Automated analysis of equivalence properties for cryptographic protocols

Remy Chretien

NNT : 2016SACLN008

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PARIS-SACLAY,
préparée à l'École Normale Supérieure de Cachan

ÉCOLE DOCTORALE N°580
Sciences et Technologies de l'Information et de la Communication

Spécialité de doctorat : Informatique

Par

Monsieur Rémy Chrétien

Analyse automatique de propriétés d'équivalence pour les protocoles cryptographiques

Thèse soutenue à Cachan, le 11 janvier 2016 :

Composition du Jury :

M David Basin	Professeur, ETH Zürich	Rapporteur
M Bruno Blanchet	Directeur de Recherche, INRIA	Rapporteur
Mme Véronique Cortier	Directrice de Recherche, CNRS	Directrice de thèse
Mme Stéphanie Delaune	Chargée de Recherche, CNRS	Directrice de thèse
M Thomas Jensen	Directeur de Recherche, INRIA	Président du Jury
M David Pointcheval	Directeur de Recherche, CNRS	Examinateur
M Géraud Sénizergues	Professeur, Université Bordeaux	Examinateur
M Dominique Unruh	Professeur, University of Tartu	Examinateur

Laboratoire Spécification et Vérification
École Normale Supérieure de Cachan, UMR 8643 du CNRS
61 avenue du Président Wilson,
94235 Cachan Cedex, France

Résumé

À mesure que le nombre d'objets capables de communiquer croît, le besoin de sécuriser leurs interactions également. La conception des protocoles cryptographiques nécessaires pour cela est une tâche notoirement complexe et fréquemment sujette aux erreurs humaines. La vérification formelle de protocoles entend offrir des méthodes automatiques et exactes pour s'assurer de leur sécurité.

Nous nous intéressons en particulier aux méthodes de vérification automatique des propriétés d'équivalence pour de tels protocoles dans le modèle symbolique et pour un nombre non borné de sessions. Les propriétés d'équivalence sont naturellement employées pour s'assurer, par exemple, de l'anonymat du vote électronique ou de la non-traçabilité des passeports électroniques.

Parce que la vérification de propriétés d'équivalence est un problème complexe, nous proposons dans un premier temps deux méthodes pour en simplifier la vérification : tout d'abord une méthode pour supprimer l'utilisation des nonces dans un protocole tout en préservant la correction de la vérification automatique; puis nous démontrons un résultat de typage qui permet de restreindre l'espace de recherche d'attaques sans pour autant affecter le pouvoir de l'attaquant.

Dans un second temps nous exposons trois classes de protocoles pour lesquelles la vérification de l'équivalence dans le modèle symbolique est décidable. Ces classes bénéficient des méthodes de simplification présentées plus tôt et permettent d'étudier automatiquement des protocoles taggués, avec ou sans nonces, ou encore des protocoles ping-pong.

Acknowledgements

I would first like to thank my advisors, Véronique Cortier and Stéphanie Delaune, for offering me such a wonderful opportunity. These years have been excellent and working with you was a real pleasure.

I am very grateful to David Basin and Bruno Blanchet who kindly accepted to review this manuscript. I also thank Thomas Jensen, David Pointcheval and Dominique Unruh and who accepted to be part of the jury. I would also like to thank Géraud Sénizergues for his collaboration and presence for my defense.

I also thank all the members of LSV for making this thesis such an enjoyable experience. Thank you for the stimulating discussions and the help throughout these years. Thank you in particular to the PhD students for making the laboratory far more than just a work place.

I want to give special thanks to all my family, who has always supported me and believed in me. Thank you for blessing me with such a nurturing home and helping me become a better person.

Finally, I would like to thank Marie for her unwavering support. Thank you for brightening up my days, even when your own plate was full too. Thank you so much.

Contents

Chapter 1

Introduction

1.1 Protocols and attacks

As technology becomes ubiquitous and software is embedded into each and every device around us, the amount of communication among connected objects and computers grows substantially. Communication protocols are designed to govern these communications. These are small programs whose goal is to allow a number of agents to transmit information between each other. They define sets of rules to properly communicate, such as for instance the format of messages that agents send and receive, their timings or error messages. Examples of protocols can be found within the Internet Protocol suite, with HTTP, TCP, UDP and many others. Protocols also appear when using RFID tags and readers to transmit data about the nature or price of a product. *Cryptographic protocols* consist of a subset of communication protocols using cryptography to achieve their objective. Cryptographic primitives like symmetric or asymmetric encryption, hash functions or zero-knowledge proofs can indeed enable properties such as secret data transmission or the establishment of secret keys for further applications. These cryptographic protocols can be found in a large number of applications. Credit cards, with or without contact, need to ensure that a payment can legitimately be made by checking the credentials and the account balance of its user. E-passports convey RFID chips which transmit to an authenticated reader, *e.g.* at a border checkpoint, the identification data (name, nationality, digitised picture) of its owner to facilitate the verification of her identity. Electronic voting, either over the internet or via electronic voting machines, make heavy use of cryptography to ensure the anonymity of voters and the validity of the ballot result.

As protocols, cryptographic or not, have to perform a large variety of tasks, we need to specify the properties they provide to their users. In the case of cryptographic protocols, we will be naturally interested in *security properties*. One of the simplest and most natural security property is the notion of secrecy: if two agents are transmitting data over some channel, can an attacker learn part or all of this data? In the credit card example, one can hope the banking account information is kept secret to avoid later frauds. Another property is authentication, describing the ability for agents to be convinced they are interacting with one another, and in the case of the credit card reader, that the card is an authentic one. For the e-passport, untraceability of a user, the impossibility for an attacker to relate two different sessions of the protocol to the same user, is often seen as desirable for the sake of privacy. In the case of electronic voting, anonymity of the voters and the verifiability of the results are two fundamental features of a good design. But to speak more precisely about security properties, we need both to make clear what our threat model is, *i.e.* what are the abilities of the attacker and the features of the network and agents, as well as explicitly state our definitions for such properties. The former will be considered in Section 1.2 and the latter in Section 1.3.

Unfortunately, even with appropriate definitions, the secure design of cryptographic protocols is a difficult task. The Needham-Schroeder protocol [58] is an authentication protocol between two agents using asymmetric cryptography and designed in 1978. Seventeen years later, Lowe discovered an attack [53] allowing a malicious agent on the network to impersonate an honest participant. More recently, an attack on the Google Single Sign-On protocol

was found by Armando *et al.* [7] in 2008. The original protocol can be schematically described in the following way, where A denotes an agent willing to authenticate to an Identity Provider I_P to gain access to the services offered by the Service Provider S_P:

$$\begin{array}{lcll} A & \rightarrow & S_P & : A, S_P, \mathsf{url} \\ S_P & \rightarrow & A & : A, I_P, \mathsf{AuthReq}(id, S_P), \mathsf{url} \\ A & \rightarrow & I_P & : A, I_P, \mathsf{AuthReq}(id, S_P), \mathsf{url} \\ I_P & \rightarrow & A & : \mathsf{sign}(\mathsf{AuthAssert}(A, I_P), \mathsf{sk}_{I_P}), \mathsf{url} \\ A & \rightarrow & S_P & : \mathsf{sign}(\mathsf{AuthAssert}(A, I_P), \mathsf{sk}_{I_P}), \mathsf{url} \end{array}$$

In this informal specification, url denotes the address of the resource A is trying to access from S_P, id is a unique identifier generated by S_P. $\mathsf{AuthReq}$ builds an authentication request for I_P to process, to which I_P answers using the authentication assertion $\mathsf{AuthAssert}$, itself signed with the private key sk_{I_P} of I_P. Note that the inner working of $\mathsf{AuthReq}$ and $\mathsf{AuthAssert}$ are not important to the attack itself, and one can just assume these to be secure primitives which guarantee authentication in a successful execution of the protocol. The core of the attack instead relies on the fact that the authentication assertion does not actually depend on url, id and S_P: the target address from A, the unique identifier id generated by S_P and the identity of the service provider S_P are not taken into account when building the authentication token $\mathsf{sign}(\mathsf{AuthAssert}(A, I_P), \mathsf{sk}_{I_P})$ that A sends to S_P in her last message to gain access to her resources. Intuitively, if able to recover such a token $\mathsf{sign}(\mathsf{AuthAssert}(A, I_P), \mathsf{sk}_{I_P})$ from A, an attacker could then use it to impersonate A and gain access to any of her resources, for any service provider using the same identity provider. This design flaw naturally leads to a man-in-the-middle attack where an attacker I, posing as a service provider and able to make A start a session with her, can gain access to this token, and hence authenticate as A to any service provider.

$$\begin{array}{lcll} I(A) & \rightarrow & S_P & : A, S_P, \mathsf{url} \\ S_P & \rightarrow & I(A) & : A, I_P, \mathsf{AuthReq}(id, S_P), \mathsf{url} \\ I(A) & \rightarrow & S_P & : \mathsf{sign}(\mathsf{AuthAssert}(A, I_P), \mathsf{sk}_{I_P}), \mathsf{url} \end{array}$$

Here $I(A)$ denotes the intruder when impersonating agent A. In this attack, we assume I got a copy of the token $\mathsf{sign}(\mathsf{AuthAssert}(A, I_P), \mathsf{sk}_{I_P})$ in a previous session, for instance by luring A into connecting to a service she controls. Then I starts a session with S_P impersonating A to gain access to authenticate and gain access to url. Instead of forwarding the authentication request of S_P to I_P ($\mathsf{AuthReq}(id, S_P)$), the attacker just uses the authentication token she already possess, along with the address of the new resource url.

This attack is actually reminiscent of the earlier attack on the Needham-Schroeder protocol, and other similar high-level attacks can also be found for other applications such as the French version of the electronic passport prior to 2010. These attacks on deployed systems highlight how error-prone the design and analysis of security protocols is. To address this difficulty, automated tools for the verification of security properties of cryptographic protocols can be designed to automatically find attacks on protocols or prove their security within a particular model.

1.2 Security and the symbolic model

In order to properly specify the security goals for our protocols, and hope to prevent attacks that would compromise these goals, we need to define a threat model. In particular, this model must explicit what are the precise capabilities of the attacker, what she can learn, what she controls and what are her objectives. The choice of this model is highly dependent on the nature of attacks we are interested in and the trust we have in the components of the system. A classical way of classifying attacks, and therefore the type of attacker, is the following one. We can first consider computational attacks on the cryptography itself: by finding flaws in the algorithms used by the cryptographic primitives, from the encryption algorithms to the pseudo-random number generators, an attacker could be able to, partially or totally, nullify the security brought by these primitives. In another type of attacks, focus is put on the low-level details of the security system. Errors or deviations from the specifications, for instance the seeding of a pseudo-random number generator

of the Debian OpenSSL package in May 2008, are found in the implementation of the protocol, leading to attacks on the program itself, rather than the specification of the protocol. A third way of modelling attacks, and the one we will be the most interested in, is to focus on the high-level specification of the protocol. In this case, the attacker is looking for logical flaws, related to the formats of the messages, leading for instance to man-in-the-middle attacks. Finally, we may consider side-channel attacks: the attacker then relies on information which was not modelled in any of the previous steps. From the power consumption of a device running the protocol to the timing used to compute some cryptographic primitives, the attacker can gain an advantage and ruin the security goals of the protocol.

In this thesis, we consider a high-level view of the cryptographic protocol. More specifically, we focus here on the symbolic Dolev-Yao model [43]. In this model, cryptographic primitives are modelled as abstract function symbols, and operations such as encryption or decryption as applications of rewriting rules or equalities modulo an equational theory. Messages are themselves built on a term algebra and exchanged on the network, which is under the control of the attacker. Even though it assumes perfect cryptography, this model gives a lot of power to the attacker and is effective in finding logical attacks in the specification of many cryptographic protocols. Flaws mentioned in Section 1.1 indeed fall into this category. This model is moreover expressive enough to express a large variety of security properties which we will classify in the next section.

1.3 Security properties

Symbolic models allow many security properties to be expressed. To do so, we can rely on the notion of knowledge of the attacker and her deduction powers. Intuitively, an attacker knowing both the encryption of a plaintext by some public key and the associated private key would be able to deduce the plaintext. Here we informally state a number of security properties.

- *Secrecy* of a message, a key or some secret data, can be stated as the inability for the attacker and for any execution of the protocol, to deduce this message.
- *Authentication* states that, for every execution of the protocol, any honest agent believing it is finishing a session with another honest agent should indeed be finishing a session with this agent. Or, in other words, no attacker is impersonating an honest agent.
- *Temporary secrecy* states that if an information is secret at some point in an execution, it will remain secret at any point in the future. In particular, all temporary secrets are permanent secrets.

Secrecy, temporary or not, and authentication all belong to a class of security properties that can be stated as follows: for any execution (also called trace) of the protocol, a property on the execution must hold. These properties are commonly referred as *trace properties* or *reachability properties*. But some properties cannot be expressed along this line. Consider the case of the anonymity of the user of some anonymous application. The possible values for the identity of the user may already be known to the attacker, as the list of user names may be public information. Unknown to her, on the other hand, are the identities of the users for a specific session.

- *Anonymity* of the said user identity can thus be modelled as the inability for the attacker to distinguish between a situation where an agent is using some identity id_1 and a situation where the same agent would be using identity id_2. Slightly more formally, if we let P be a formal specification of the anonymous application under investigation, $P[^{id_1}/_{id}]$ the variant of P where our agent's identity is id_1 and $P[^{id_2}/_{id}]$ when the value is id_2, the anonymity of the identity can be stated as $P[^{id_1}/_{id}] \approx P[^{id_2}/_{id}]$ where $\approx$ denotes some equivalence relation between protocols, representing the inability for the attacker to distinguish between two protocols.
- *Untraceability* is another property related to the ability for an attacker to link together messages from the same execution of a protocol. It models an attacker trying to trace an agent by trying to recognise the sessions which generated particular messages. It is a desirable property in the case of e-passports, where an attacker should not

be able to link messages together and thus track the movements of its bearer. More formally, for untraceability to hold, an attacker should think than each session she observes has been generated by a new distinct e-passport. If P_{multi} is the original version of the e-passport protocol where each passport can take part into any number of sessions, and P_{single} is an ideal version where each passport can only execute one session, then the equivalence $P_{\mathsf{multi}} \approx P_{\mathsf{single}}$ should hold.

An alternative interpretation of these equivalences is the inability for an attacker, given two protocols, to accurately guess for any execution (or trace) of one of those two, which protocol produced it. Properties which can be expressed as such are usually referred as *equivalence properties*. Most properties relevant to privacy are actually modelled as equivalences, *e.g.* in voting protocols, as well as more exotic properties, for example in mobile ad-hoc routing [28].

1.4 Automation of security proofs

Abstraction of the messages and the underlying cryptography in the symbolic models opens up the possibility of efficiently automating the process of attack discovery and, conversely, the verification of the system security. This formalisation indeed allows to benefit from classical models and techniques in theoretical computer science, ranging from model-checking to resolution and rewriting systems. Tools, as further described in Section 1.5, are for instance able to catch the attacks mentioned in Section 1.1. Still, automatically verifying a security property remains a difficult task, whose complexity is directly related to the power which is given to the attacker as well as the intricacy of the security property considered. In the following sections, we give a panorama of these difficulties and the techniques employed to deal with different kind of properties.

1.4.1 (Un)decidability of reachability properties

We first consider reachability properties, such as secrecy and authentication, and the existing decidability or undecidability results associated. In the most general setting, reachability properties can already be shown to be undecidable [45, 57] with minimal cryptographic primitives such as symmetric encryption and pairs. A simple encoding relies on the arbitrary size of messages to store the words needed to encode the Post Correspondence Problem, encrypted to avoid unwanted alterations by the attackers. But even if the unbounded size of messages can be seen as an undesirable property, undecidability of secrecy can still be proven with messages of bounded size [4], by using nonces, fresh values generated by the protocol and attacker, as pointers to encode, once again, an instance of the Post Correspondence Problem.

For this reason, in order to achieve decidability, one needs to consider restrictions, either on the protocols or on the power given to the attacker when verifying a security property. One of the most classical ones consists in *bounding the number of sessions* in the verification. This limitation practically forbids any agent to execute a protocol more than a fixed number of times, number set *a priori*. Although this restriction manages to achieve decidability for reachability properties [55, 11, 35], it fails to be entirely acceptable. The bound on the number of sessions being arbitrary, it provides no guarantee on the security of the protocol if no attack were to be found for a given number of sessions. Moreover, existing tools are only able to handle a low number of sessions, barring the practical verification of a protocol for an increasing large number of sessions.

These obstacles are thus incentive to focus on verification for an unbounded number of sessions. Another line of restrictions in that direction is to restrict the ability of the agents to blindly copy portions of messages they received. More formally, by limiting the amount of information an agent can forward, secrecy can again be proven decidable [31]. Another way of achieving decidability is to consider tagged protocols. Tagging can be first considered from a practical implementation perspective as a way of making messages unambiguous, a good practice in protocol design [54]. By adding a specific constant, the tag, to the beginning of any encrypted plaintext, an agent can easily deduce from which stage of the protocol this message originate. For this reason, it prevents a number of type-flaw

attacks where a message being parsed as another resulted in unwanted behaviours, for instance in the Otway-Rees protocol [59]. On top of that, tagging schemes can also lead to decidability for secrecy [60] or temporary secrecy [47].

1.4.2 Proof techniques for reachability properties

As we have seen earlier, reachability properties are undecidable in general without tight restrictions. But some techniques can still be used to reduce the difficulty of the verification of reachability properties. Tagging schemes, as mentioned previously in Section 1.4.1, are also able to greatly reduce the space of traces to consider when checking for secrecy thanks to typing results [6]. Well-typed traces indeed constrain the size and shape of the messages which can be exchanged by both the honest agents and the attacker.

Another way of easing verification is to prevent agents from generating new nonces. The infinite number of nonces to consider is a difficult point to address as it leads to consider infinitely many messages of any size and causes undecidability [4] as already discussed. Fortunately, when checking for secrecy, removing nonce generation from the specification of a protocol without else branches is sound. Which is to say that if $\overline{P}$ is a version of a protocol P with its nonce generation removed and $\overline{P}$ preserves the secret of some value s, then P also preserves the secrecy of s. Even if the converse is not true in general, it still provides a safe way to prove secrecy properties for protocols with nonces when using only procedures designed for protocols without nonces. More generally, approximations on nonces enable tools such as ProVerif [13] to soundly prove security properties.

1.4.3 The case of equivalence

Verifying trace equivalence is, unfortunately, more difficult in general than verifying reachability properties. A property like secrecy can for instance very easily be encoded as an equivalence between two processes which differ only if the secret data is disclosed. Indeed, on top of the existing difficulties for reachability properties, equivalence itself adds a new layer of complexity as even when considering substantially simpler process algebra without terms to model protocols, such as CCS or the π-calculus, equivalence is known to be very hard and in most cases an undecidable problem [48].

Specific to the verification of equivalence properties, there is actually some leeway in the precise definition of the equivalence of protocols we consider. Several notions have been described in the literature, such as trace equivalence, may-testing equivalence or observational equivalence [17, 2, 1]. In this thesis we consider *trace equivalence* as our core notion of equivalence. Other notions still coexist and can provide alternate methods for checking equivalence. Labeled bisimilarity for instance offers a tighter notion of equivalence between processes, and differ from trace equivalence in the same way as for process algebra without terms, which is in particular helpful in providing proofs of equivalence [65], instead of providing attacks *i.e.* witnesses of non-equivalence, as is usually done by tools. Diff-equivalence [14] offers another example of a tighter notion of equivalence whose verification can be encoded within the tool ProVerif and leads to a sound non-terminating procedure for checking equivalence, albeit prone to false negatives. Nevertheless, when dealing with "deterministic protocols" such as determinate protocols, a number of these equivalences actually collapse [19].

As mentioned before, verifying equivalence is a difficult problem and there exist very few decidability results. In particular, they all consider equivalence for a bounded number of sessions [12, 35] and are thus subject to the same limitations as discussed in Section 1.4.1. One way to limit the number of traces that can be produced in response to the actions of the attacker is to restrict the branching behaviour of protocols. The use of if/then conditional statements is invaluable for providing agents with the ability to test values, such as signatures or keys, before proceeding to the remaining of the protocol. Unfortunately, else branches in these conditional statements tend to make equivalence checking much more difficult as they require to analyse any path that can be taken through all the conditionals in an execution. For this reason, removing else is often used as a restriction. A common restriction on protocols for proving equivalence is to consider "deterministic protocols". During an execution, a protocol can be presented with non-deterministic choices. In most process algebras, when presented with a message from the network which can be

received by two agents, *e.g.* when the same channel is used, a choice must be made to decide which of the two agents should capture the message and proceed. More directly, some algebras contain a non-deterministic choice operator (usually "+") which can force an agent to make such a choice. These choices impact the number of interleavings of actions to consider, and similarly to the else branches, it is a natural hypothesis to remove any occurrences of such choices. This leads to notions of determinism, for which only one behaviour is possible in response to the given actions of the attacker. This determinism can take the form of determinate protocols [19] or simple protocols [33].

1.5 Existing tools

To fill the need for the automated verification of protocols, a number of tools were developed to find attacks and prove security properties about cryptographic protocols. Each of them relies on a number of assumptions on the protocols, properties and equivalence to consider in order to obtain sound verification procedures. In what follows, we focus on tools which are either devoted or contain features to prove the equivalence of protocols. Some are full decision procedures whereas other do not always provide termination in every case.

- *Spec* [65]: Implemented on top of the Bedwyr logical framework [9], Spec verifies open bisimulation for protocols modelled in a version of the spi-calculus. It relies on a fixed set of cryptographic primitives and only handles a bounded number of sessions, while guaranteeing termination. Thanks to the open bisimulation, the tool is able to provide proofs of equivalence, in the form of the bisimulation relation.
- *Akiss* [19]: Akiss is another tool which verifies equivalence for a bounded number of sessions. It uses an encoding as Horn clauses and focuses on trace equivalence. It does not support else branches and may not terminate but it can deal with large number of primitives and equational theories, as well as provide witness of non-equivalence.
- *Apte* [20]: Apte deals with the verification of trace equivalence for a bounded number of sessions and for standard primitives. It guarantees termination and supports both else branches and private channels.
- *ProVerif* [13]: Initially developed for reachability properties, it was extended to prove a stronger notion of equivalence: the diff-equivalence of bi-processes [14]. Like the original version, termination is not guaranteed when proving equivalence. ProVerif uses an encoding as Horn clauses and supports private channels and a large number of equational theories. Contrary to the previous tools, ProVerif consider an unbounded number of sessions but can produces false negatives, despite recent improvements [21, 40].
- *Maude-NPA* [61]: Maude-NPA is an analysis tool based on multiset-rewriting. As ProVerif, it handles an unbounded number of sessions and proves a tighter version of protocol equivalence. It also puts an emphasis on algebraic properties of primitives to deal, in particular, with Associative-Commutative equational theories, even though it currently handles rather simple protocols.
- *Tamarin* [10]: As ProVerif and Maude-NPA, Tamarin deals with an equivalence stronger than trace equivalence and can handle an unbounded number of sessions. It also uses multiset rewriting and can either produce attacks or prove equivalence, at the cost of potential non-termination, for a large number of primitives and theories. It moreover provides an interactive mode for proving properties.

1.6 Contributions

The main objective of my thesis is to prove decidability results for checking trace equivalence of protocols, as well as to provide new proof methods to ease the automated verification of protocol equivalence. My contributions can be split into five main points, which are summarised here. The two first aspects are related to ways of making equivalence

checking easier, by soundly removing some of its complexity or by sharply restricting the search spaces to consider. The last three consists of new classes of protocols such that trace equivalence are decidable. Combined with the facilitating methods described earlier, they further enlarge the classes of protocols for which equivalence properties can be proven.

1.6.1 Simplifying equivalence checking

Sound removal of nonces for checking protocol equivalence As mentioned in Section 1.4, the existence of an unbounded number of nonces that the agents and the attacker can use is an important source of complexity for the verification of security properties for an unbounded number of sessions. In the case of secrecy, this difficulty can be partially avoided by the soundness of direct removal of any nonce generation in the specification of the protocol (see Section 1.4.2), *i.e.* if no attack is found when nonces are removed, then no attack whatsoever can be found. This implication is unfortunately false in the case of equivalence. Our first contribution, presented in Chapter 3, consists then in a new transformation defined on protocols which removes nonce generation while remaining a sound abstraction for the equivalence with a unbounded number of sessions. Hence, if P and Q are two protocols, with arbitrary name generation, we algorithmically define two transformed versions of these, $\overline{P}$ and $\overline{Q}$ *without nonces*, such that $\overline{P} \approx \overline{Q} \Rightarrow P \approx Q$. This transformation works for a large number of cryptographic primitives and rewriting systems, namely *adequate theories*, which encompass traditional symmetric or asymmetric encryption, signatures or zero-knowledge proofs and *simple protocols*. Compared to the naive nonce deletion used to deal with secrecy, this transformation considers an additional copy of each nonce before deleting it. This result allows to directly turn a decision procedure for trace equivalence of protocols *without nonces* but for an unbounded number of sessions, such as the one presented in Chapter 5, into a terminating and sound procedure to verify trace equivalence of protocols *with nonces* and for an unbounded number of sessions.

Search space restriction through typing Section 1.4 described the use of tagging as a good and reasonable design practice. It also introduced the role of typing in the decidability results obtained for the weak secrecy within tagged protocols. Our second contribution, presented in Chapter 4 offers a generalisation of both tagging and typing to the case of trace equivalence for *determinate protocols* and an unbounded number of sessions. While focusing on the theory of symmetric encryption, it offers an important reduction in the set of traces searched for proving equivalence. It happens to be instrumental for the decidability results presented in Chapters 5 and 6. Intuitively, the notion of *type-compliance*, introduced to generalise the idea of tagging, constrains protocols to avoid ambiguity in the interpretation of the origin of any message sent on the network and therefore drastically limit the shape and size of messages which can be exchanged between honest agents. The typing result ensures that limiting the attacker to abide by these constrained shapes and sizes of messages does not reduce her power at all. More formally, we prove that if an attacker is able to produce a witness of non-equivalence between two type-compliant protocols, then she is also able to produce a *well-typed* witness of non-equivalence between those two protocols, *i.e.* an attack such that any message sent on the network by the agents and the attacker fits a particular format defined by the specification of the protocol. To achieve this result, we designed a decision procedure for trace equivalence with a bounded number of sessions which produces, in case of non-equivalence, only well-typed witnesses.

1.6.2 Decidable classes

Decidability of equivalence for simple protocols without nonces Decidability results for equivalence for an unbounded number of sessions are difficult to obtain, due to the inherent complexity of protocol equivalence as discussed in Section 1.4. Chapter 5 presents one of the first decidability results for trace equivalence between simple type-compliant protocols without nonces. This result deeply relies on the typing theorem from Chapter 4 to bound the search space for attacks with such protocols. The additional restriction on nonces allows us to further bound the total number of messages which can be exchanged on the network, either by honest agents or the attacker. This bound can

then be translated into an upper bound on the length of any minimal witness of non-equivalence, if it exists. Moreover, even though it does not deal directly with nonces, this class is compatible with the abstraction presented in Chapter 3 to produce a sound procedure for checking equivalence for simple type-compliant protocols with nonces.

Decidability of equivalence for tagged protocols with nonces Chapter 6 introduces the first result of decidability for an unbounded number of sessions and tagged protocols with nonces. The typing result introduced in Chapter 4 offered a reduction in the search space when proving equivalence. This work pursues it by refining the typing systems considered to obtain *structure-preserving typing systems*, a notion closely related to tagging schemes in the literature. It also develops the notion, independently presented in [47] for the restricted case of temporary secrecy, of *dependency graph* which formally abstracts the dependencies between the actions in a protocol specification. It allows to define *acyclic protocols*, intuitively related to protocols without loops in their natural executions. Focusing on simple acyclic type-compliant protocols using symmetric encryption, we provide a decidability result for an unbounded number of sessions as well as an unbounded number of nonces. This result also provides a computable upper bound on the length of any potential minimal witness of non-equivalence between such protocols.

Decidability of equivalence for ping-pong protocols The last contribution of this thesis, detailed in Chapter 7, presents another decidability result for trace equivalence of protocols for an unbounded number of sessions. It focuses on a class of *ping-pong protocols*, reminiscent of the class studied in the seminal work of Dolev and Yao [43] with cryptographic primitives such as symmetric and asymmetric encryption, signature and hash functions, but without pairing or nonces. It provides a two-way reduction between trace equivalence of ping-pong protocols and language equivalence of deterministic pushdown automata [63] as well as a decidability result for ping-pong protocols for an unbounded number of sessions. As a side result, it also produces a new undecidability result for trace equivalence with very elementary protocols too, further highlighting the difficulty of this task. Based on the complexity of the reductions between protocols and deterministic pushdown automata [62, 64], it also sheds some light on the inherent complexity of trace equivalence. Focusing on trace equivalence for an unbounded number of sessions without nonces, this result appears to be a prime candidate for the application of the abstraction discussed in Chapter 3.

1.7 Outline

The remainder of this thesis is articulated into two main parts, focusing first on easing the verification of trace equivalence mainly through nonce removal and typing and then on describing classes of protocols for which trace equivalence is decidable.

Part I is made of three chapters. Chapter 2 aims at defining a unified formal model for protocols and equivalence checking which will be used throughout this thesis, facilitating the application of the subsequent results. Following that, Chapter 3 proposes a first method to simplify equivalence checking by removing nonces and effectively reducing the equivalence checking of protocols with an unbounded number of nonces to protocols with a bounded number of nonces. Chapter 4 then proposes a new practical restriction on protocols and executions in order to reduce equivalence checking to the verification of *well-typed* equivalence.

Part II compiles three classes of protocols for which trace equivalence can be proven decidable with an unbounded number of sessions. Chapter 5 describes a decidable class of protocols with a bounded number of nonces naturally formed from the restrictions introduced in Chapter 4 and which easily interfaces itself with the result from Chapter 3. Then Chapter 6 introduces a novel notion of *acyclic protocols* and *dependency graph* to define a practical class of protocols with an unbounded number of nonces and sessions for which equivalence is decidable. Finally, Chapter 7 describes a final class of protocols, *ping-pong protocols*, whose equivalence is reduced to the equivalence of language between deterministic pushdown automata and hence decidable.

Appendix A and B ultimately contain proofs from Chapters 4 and 7, respectively, which were not included into the body of this thesis.

1.8 Publications

- The results from Chapter 3 led to a conference article presented at the 20th European Symposium On Research In Computer Science (ESORICS) in 2015 [25].
- The results from Chapter 4 and 5 led to a conference article presented at the 25th Conference on Concurrency Theory (CONCUR) in 2014 [24].
- The results from Chapter 7 led to a conference article presented at the 40th International Colloquium on Automata, Languages and Programming (ICALP) in 2013 [23], as well as a journal version in Transactions on Computational Logic in 2015 [27].
- The results from Chapter 6 finally led to a conference article presented at the 28th IEEE Computer Security Foundations Symposium (CSF) in 2015 [26].

Part I

Simplifying equivalence checking

Chapter 2

Model for general protocols

In this chapter, we present the model we will use through this thesis to model protocols and their properties. Security protocols are modelled through a process algebra inspired from the applied pi-caculus [1] that manipulates terms. Still, we consider a variant with several differences: we do not consider else branches in conditional expressions, nor do we allow private channels. We also introduce several classes of protocols which convey the idea of determinism as mentioned in introduction: *determinate* and *simple* protocols. Section 2.5 finally describes the core notion of *type-compliant* protocols, related to the tagging introduced in [15]. This idea on protocols will constitute the central hypothesis of the results of Chapters 4 and 6.

2.1 Term algebra

We assume an infinite set $\mathcal{N}$ of *names*, which are used to represent keys and nonces and an infinite set $\mathcal{X}$ of variables. We assume a signature Σ, *i.e.* a set of function symbols together with their arity, and we make a distinction between *constructor* symbols and *destructor* symbols: $\Sigma = \Sigma_c \uplus \Sigma_d$. Given a signature Σ, we denote by $\mathcal{T}(\Sigma, \mathsf{A})$ the set of terms built from symbols in Σ and atomic data in A. Terms without variables are called *ground*. The set $\mathcal{T}(\Sigma_c, \mathcal{X} \cup \mathcal{N})$ is the set of *constructor terms*. Then among the terms in $\mathcal{T}(\Sigma_c, \mathcal{N})$ we distinguish a special subset of terms called *messages* and noted $\mathcal{M}_\Sigma$, and that is stable under renaming of names: a message does *not* contain any destructor symbol, and $m \in \mathcal{M}_\Sigma$ implies that $m\rho \in \mathcal{M}_\Sigma$ for any renaming ρ (not necessarily a bijective one).

In addition to the set of variables $\mathcal{X}$, we consider an infinite disjoint set of variables $\mathcal{W}$. Variables in $\mathcal{W}$ intuitively refer to variables used to store messages learnt by the attacker. We denote $vars(u)$ the set of variables that occur in a term u. The application of a substitution σ to a term u is written $u\sigma$, and we denote $dom(\sigma)$ its *domain*. The *positions* of a term are defined as usual. Two terms u and v are *unifiable* if there is a substitution σ such that $u\sigma = v\sigma$.

The properties of the primitives are expressed using rewriting rules of the form $\mathsf{g}(t_1, \ldots, t_n) \to t$ where g is a destructor, that is $\mathsf{g} \in \Sigma_d$, and $t_1, \ldots, t_n, t$ are constructor terms. A rewriting rule can only be applied to constructor terms. Formally, we say that u can be *rewritten into* v if there is a position p and a rule $\mathsf{g}(t_1, \ldots, t_n) \to t$ such that u at position p is equal to $\mathsf{g}(t_1, \ldots, t_n)\theta$ and $v = u[t\theta]_p$ (that is u where the term at position p has been replaced by $t\theta$) for some substitution θ such that $t_1\theta, \ldots, t_n\theta, t\theta$ are messages. We only consider sets of rewriting rules that yield convergent rewrite systems. We denote by $u{\downarrow}$ the *normal form* of a given term u. We refer the reader to [42] for the precise definitions of rewriting systems, convergence, and normal forms.

Example 2.1.1. A typical signature for representing symmetric encryption and pair is

$$\Sigma = \{\mathsf{senc},\ \mathsf{sdec},\ \langle\ \rangle,\ \mathsf{proj}_1, \mathsf{proj}_2\} \uplus \Sigma_0$$

where Σ_0 is a set of atomic data. The set Σ_0 typically contains the public constants known to the attacker (*e.g.* agent names a, b, ...). The symbols senc and sdec of arity 2 represent symmetric encryption and decryption. Pairing

is modelled using $\langle\ \rangle$ of arity 2, whereas projection functions are denoted proj_1 and proj_2 (both of arity 1). The relations between encryption/decryption and pairing/projections are represented through the following convergent rewrite system:

$$\mathsf{sdec}(\mathsf{senc}(x, y), y) \to x, \quad \text{and} \quad \mathsf{proj}_i(\langle x_1, x_2\rangle) \to x_i \text{ with } i \in \{1, 2\}.$$

We have that $\mathsf{proj}_1(\mathsf{sdec}(\mathsf{senc}(\langle s_1, s_2\rangle, k), k))\downarrow = s_1$. Note that, since a destructor can only be applied on messages, no rewriting rule can be applied on the term $\mathsf{sdec}(\mathsf{senc}(s, \mathsf{proj}_1(s)), \mathsf{proj}_1(s))$ which is thus in normal form (but not a message). This signature Σ is split into two parts as follows: $\Sigma_c = \{\mathsf{senc}, \langle\ \rangle\} \uplus \Sigma_0$ and $\Sigma_d = \{\mathsf{sdec}, \mathsf{proj}_1, \mathsf{proj}_2\}$. Then, we may consider $\mathcal{M}_\Sigma$ to be $\mathcal{M}_c = \mathcal{T}(\Sigma_c, \mathcal{N})$ the set of all ground constructor terms. We may also restrict $\mathcal{M}_\Sigma$ to be $\mathcal{M}_{\mathsf{atomic}}$, the set of ground constructor terms that only use atomic data in key position.

Finally, we assume Σ to be split into two parts, and this distinction is orthogonal the one made between destructor and constructor symbols. We denote by Σ_{pub} the set of function symbols that are public, *i.e.* available to the attacker, and Σ_{priv} for those that are private. Actually, an attacker builds his own messages by applying public function symbols to terms he already knows. Formally, a computation done by the attacker is modelled by a term in $\mathcal{T}(\Sigma_{\mathsf{pub}}, \Sigma_0 \cup \mathcal{W})$, called a *recipe*. Note that such a term does *not* contain any name. Indeed, all names are initially unknown to the attacker.

2.2 Process algebra

Let $\mathcal{C}h$ be an infinite set of *channels*. We consider processes built using the grammar below where $u \in \mathcal{T}(\Sigma_c, \mathcal{N} \cup \mathcal{X})$, $v \in \mathcal{T}(\Sigma, \mathcal{N} \cup \mathcal{X})$, $n \in \mathcal{N}$, and $c, c' \in \mathcal{C}h$:

P, Q	:=	0	*null*
	|	$\mathsf{in}(c, u).P$	*input*
	|	$\mathsf{out}(c, u).P$	*output*
	|	$\mathsf{let}\ x = v\ \mathsf{in}\ P$	*evaluation*
	|	$(P \mid Q)$	*parallel*
	|	$!P$	*replication*
	|	$\mathsf{new}\ n.P$	*restriction*
	|	$\mathsf{new}\ c'.\mathsf{out}(c, c').P$	*channel generation*

The process 0 does nothing. The process "$\mathsf{in}(c, u).P$" expects a message m of the form u on channel c and then behaves like $P\sigma$ where σ is a substitution such that $m = u\sigma$. The process "$\mathsf{out}(c, u).P$" emits u on channel c, and then behaves like P. The variables that occur in u are instantiated when the evaluation takes place. The process "$\mathsf{let}\ x = v\ \mathsf{in}\ P$" tries to evaluate v and in case of success the process P is executed; otherwise the process is blocked. The process "$P \mid Q$" runs P and Q in parallel. The process "$!P$" executes P some arbitrary number of times. The restriction "$\mathsf{new}\ n$" is used to model the creation of a fresh random number (*e.g.*, a nonce or a key) whereas channel generation "$\mathsf{new}\ c'.\mathsf{out}(c, c').P$" is used to model the creation of a fresh channel name that shall immediately be made public. Note that we consider only public channels. It is still useful to generate fresh channel names to let the attacker identify the different sessions (as it is often the case in practice through sessions identifiers).

Note that our calculus allows both message filtering as well as explicit application of destructor symbols. For example, to represent a process that waits for a message, decrypts it with a key k, and sends the plaintext in clear, we may write $P = \mathsf{in}(c, \mathsf{senc}(x, k)).\mathsf{out}(c, x)$ as well as $Q = \mathsf{in}(c, y).\mathsf{let}\ x = \mathsf{sdec}(y, k)\ \mathsf{in}\ \mathsf{out}(c, x)$. However, the choice of filtering or let yields a slightly different behaviour since a message will be received in P only if it matches the expected format while any message will be received in Q (and then the format is checked).

We write $fv(P)$ for the set of *free variables* that occur in P, *i.e.* the set of variables that are not in the scope of an input or a let construction. We assume $\mathcal{C}h = \mathcal{C}h_0 \uplus \mathcal{C}h^{\mathsf{fresh}}$ where $\mathcal{C}h_0$ and $\mathcal{C}h^{\mathsf{fresh}}$ are two infinite sets of channels. Intuitively, channels of $\mathcal{C}h^{\mathsf{fresh}}$, denoted $ch_1, \ldots, ch_i, \ldots$ will be used in the semantics to *instantiate* the channels generated during the execution of a protocol. They shall not be part of its specification.

Definition 2.2.1. A *protocol* P is a process such that P is ground, *i.e.* $fv(P) = \emptyset$; and P does not use channel names from $\mathcal{C}h^{\mathsf{fresh}}$.

Example 2.2.1. The Denning Sacco protocol [30] (without timestamps) is a key distribution protocol using symmetric encryption and a trusted server. It can be described informally as follows:

$$\begin{array}{l} 1.\ A \to S:\ A, B \\ 2.\ S \to A:\ \{B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}} \\ 3.\ A \to B:\ \{K_{ab}, A\}_{K_{bs}} \end{array}$$

where $\{m\}_k$ denotes the symmetric encryption of a message m with key k. The agents A and B aim at authenticating each other and establishing a session key K_{ab} through a trusted server S. The key K_{as} (resp. K_{bs}) is a long term key shared between A and S (resp. B and S).

We model the Denning Sacco protocol in our formalism. Below, k_{as}, k_{bs}, k_{ab} are names, whereas a and b are constants from Σ_0. We denote by $\langle x_1, \ldots, x_{n-1}, x_n\rangle$ the term $\langle x_1, \langle \ldots \langle x_{n-1}, x_n\rangle\rangle\rangle$. The protocol is modelled by the parallel composition of three processes P_A, P_B, and P_S, corresponding to the roles of A, B, and S.

$$\begin{array}{rl} P_{\mathsf{DS}} = & !\,\mathsf{new}\ c_1.\mathsf{out}(c_A, c_1).P_A \ \mid !\,\mathsf{new}\ c_2.\mathsf{out}(c_B, c_2).P_B \\ & \mid !\,\mathsf{new}\ c_3.\mathsf{out}(c_S, c_3).P_S \end{array}$$

The processes P_A, P_B, and P_S are given below.

$$\begin{array}{rcl} P_A & = & \mathsf{out}(c_1, \langle \mathsf{a}, \mathsf{b}\rangle). \\ & & \mathsf{in}(c_1, \mathsf{enc}(\langle \mathsf{b}, x_{AB}, x_B\rangle, k_{as})). \\ & & \mathsf{out}(c_1, x_B) \\ P_B & = & \mathsf{in}(c_2, \mathsf{enc}(\langle y_{AB}, \mathsf{a}\rangle, k_{bs})) \\ P_S & = & \mathsf{in}(c_3, \langle \mathsf{a}, \mathsf{b}\rangle).\ \mathsf{new}\ k_{ab}. \\ & & \mathsf{out}(c_3, \mathsf{enc}(\langle \mathsf{b}, k_{ab}, \mathsf{enc}(\langle k_{ab}, \mathsf{a}\rangle, k_{bs})\rangle, k_{as})) \end{array}$$

2.3 Semantics

The operational semantics of a process is defined using a relation over configurations. A *configuration* is a pair $(\mathcal{P}; \phi)$ where:

- $\mathcal{P}$ is a multiset of ground processes.
- $\phi = \{\mathsf{w}_1 \ \triangleright\ m_1, \ldots, \mathsf{w}_n \ \triangleright\ m_n\}$ is a *frame*, *i.e.* a substitution where $\mathsf{w}_1, \ldots, \mathsf{w}_n$ are variables in $\mathcal{W}$, and $m_1, \ldots, m_n$ are messages, *i.e.* terms in $\mathcal{M}_\Sigma$.

We often write P instead of $(\{P\}; \emptyset)$, and $P \cup \mathcal{P}$ or $P \mid \mathcal{P}$ instead of $\{P\} \cup \mathcal{P}$. The terms in ϕ represent the messages that are known by the attacker.

The frame ϕ represents the messages the attacker has learnt so far. He may deduce new messages from his knowledge. This is formalised through the deducibility notion.

Definition 2.3.1. A message u is *deducible* from a frame ϕ, denoted $\phi \vdash u$, if there exists a recipe R such that $R\phi{\downarrow} = u$.

The operational semantics of a process is induced by the relation $\xrightarrow{\alpha}$ as defined in Figure 2.1.

The first rule allows the attacker to send to some process a term built from publicly available terms and symbols. The second rule corresponds to the output of a term: the corresponding term is added to the frame of the current configuration, which means that the attacker can now access the sent term. Note that the term is outputted provided

$$(\mathsf{in}(c,u).P \cup \mathcal{P};\phi) \xrightarrow{\mathsf{in}(c,R)} (P\sigma \cup \mathcal{P};\phi) \qquad \text{where } R \text{ is a recipe such that } R\phi\!\downarrow \text{ is a message and } R\phi\!\downarrow = u\sigma \text{ for some } \sigma \text{ with } dom(\sigma) = vars(u)$$

$$(\mathsf{out}(c,u).P \cup \mathcal{P};\phi) \xrightarrow{\mathsf{out}(c,\mathsf{w}_{i+1})} (P \cup \mathcal{P};\phi \cup \{\mathsf{w}_{i+1} \triangleright u\}) \qquad \text{where } u \text{ is a message and } i \text{ is the number of elements in } \phi$$

$$(\mathsf{new}\, c'.\mathsf{out}(c,c').P \cup \mathcal{P};\phi) \xrightarrow{\mathsf{out}(c,ch_i)} (P\{^{ch_i}/_{c'}\} \cup \mathcal{P};\phi) \qquad \text{where } ch_i \text{ is the "next" fresh channel name available in } \mathcal{C}h^{\mathsf{fresh}}$$

$$(\mathsf{let}\ x = v\ \mathsf{in}\ P \cup \mathcal{P};\phi) \xrightarrow{\tau} (P\{^{v\downarrow}/_{x}\} \cup \mathcal{P}\ \phi) \qquad \text{where } v\!\downarrow \text{ is a message}$$

$$(\mathsf{new}\ n.P \cup \mathcal{P};\phi) \xrightarrow{\tau} (P\{^{n'}/_{n}\} \cup \mathcal{P};\phi) \qquad \text{where } n' \text{ is a fresh name in } \mathcal{N}$$

$$(!P \cup \mathcal{P};\phi) \xrightarrow{\tau} (P \cup\, !P \cup \mathcal{P};\phi)$$

Figure 2.1: Semantics of protocols

that it is a message. The third rule corresponds to the special case of an output of a freshly generated channel name. In such a case, the channel is not added to the frame but it is implicitly assumed known to the attacker, as all the channel names. These three rules are the only observable actions. The fourth rule corresponds to the evaluation of the term v; if this succeeds, *i.e.* if $v\!\downarrow$ is a message then x is bound to the result and P is executed; otherwise the process is blocked. The two remaining rules are quite standard and are unobservable by the attacker.

The relation $\xrightarrow{\alpha_1 \ldots \alpha_n}$ between configurations (where $\alpha_1 \ldots \alpha_n$ is a sequence of actions) is defined as the transitive closure of $\xrightarrow{\alpha}$. Given a sequence of observable actions tr, we write $K \overset{\mathsf{tr}}{\Longrightarrow} K'$ when there exists a sequence $\alpha_1 \ldots \alpha_n$ such that $K \xrightarrow{\alpha_1 \ldots \alpha_n} K'$ and tr is obtained from $\alpha_1 \ldots \alpha_n$ by erasing all occurrences of τ. For every protocol P, we define its *set of traces* as follows:

$$\mathsf{trace}(P) = \{(\mathsf{tr},\phi) \mid P \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P};\phi) \text{ for some configuration } (\mathcal{P};\phi)\}.$$

Example 2.3.1. From Example 2.2.1, consider the following sequence tr:

$$\begin{aligned}\mathsf{tr} = \quad & \mathsf{out}(c_A, ch_1).\mathsf{out}(c_B, ch_2).\mathsf{out}(c_S, ch_3).\\ & \mathsf{out}(ch_1, \mathsf{w}_1).\mathsf{in}(ch_3, \mathsf{w}_1).\mathsf{out}(ch_3, \mathsf{w}_2).\\ & \mathsf{in}(ch_1, \mathsf{w}_2).\mathsf{out}(ch_1, \mathsf{w}_3).\mathsf{in}(ch_2, \mathsf{w}_3)\end{aligned}$$

This sequence tr allows one to reach the frame:

$$\phi = \quad \{\mathsf{w}_1 \triangleright \langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{w}_2 \triangleright \mathsf{enc}(\langle \mathsf{b}, k_{ab}, \mathsf{enc}(\langle k_{ab}, \mathsf{a}\rangle, k_{bs})\rangle, k_{as}), \mathsf{w}_3 \triangleright \mathsf{enc}(\langle k_{ab}, \mathsf{a}\rangle, k_{bs})\}.$$

We have that $(\mathsf{tr},\phi) \in \mathsf{trace}(P_{\mathsf{DS}})$. This trace corresponds to a normal execution of the protocol. Moreover we have that, for instance, a is deducible, as $\mathsf{proj}_1(\mathsf{w}_1)\phi\!\downarrow = \mathsf{a}$.

2.4 Trace equivalence

Intuitively, two protocols are equivalent if they cannot be distinguished by any attacker. Trace equivalence can be used to formalise many interesting security properties, in particular privacy-type properties, such as those studied for instance in [18, 39]. We first define indistinguishability of sequences of messages, called *static equivalence*.

Definition 2.4.1. Two frames ϕ_1 and ϕ_2 are *statically equivalent*, $\phi_1 \sim \phi_2$, when we have that $dom(\phi_1) = dom(\phi_2)$, and:

- for any recipe R, $R\phi_1\!\downarrow \in \mathcal{M}_\Sigma$ if, and only if, $R\phi_2\!\downarrow \in \mathcal{M}_\Sigma$; and
- for all recipes R_1 and R_2 such that $R_1\phi_1\!\downarrow, R_2\phi_1\!\downarrow \in \mathcal{M}_\Sigma$, we have that
$$R_1\phi_1\!\downarrow = R_2\phi_1\!\downarrow \text{ if, and only if, } R_1\phi_2\!\downarrow = R_2\phi_2\!\downarrow.$$

Intuitively, two frames are equivalent if an attacker cannot see the difference between the two situations they represent. If some computation fails in ϕ_1 for some recipe R, *i.e.* $R\phi_1\downarrow$ is not a message, it should fail in ϕ_2 as well. Moreover, the frames ϕ_1 and ϕ_2 should satisfy the same equalities. In other words, the ability of the attacker to distinguish whether a recipe R produces a message, or whether two recipes R_1, R_2 produce the same message should not depend on the frame.

Example 2.4.1. Consider $\phi_1 = \{\mathsf{w}_1 \triangleright \mathsf{senc}(\mathsf{m}_1, \mathsf{k}_i)\}$, and $\phi_2 = \{\mathsf{w}_1 \triangleright \mathsf{senc}(\mathsf{m}_2, \mathsf{k}_i)\}$. Assuming that m_1, m_2 and k_i are public constants from Σ_0, we have that $\phi_1 \not\sim \phi_2$. An attacker can observe that decrypting the message of ϕ_1 with the public constant k_i leads to the public constant m_1. This is not the case in ϕ_2. Consider the recipes $R_1 = \mathsf{sdec}(\mathsf{w}_1, \mathsf{k}_i)$ and $R_2 = \mathsf{m}_1$. We have that $R_1\phi_1\downarrow = R_2\phi_1\downarrow$ whereas $R_1\phi_2\downarrow \neq R_2\phi_2\downarrow$.

Intuitively, two protocols are *trace equivalent* if, however they behave, the resulting sequences of messages observed by the attacker are in static equivalence.

Definition 2.4.2. Let P and Q be two protocols. We have that $P \sqsubseteq Q$ if for every $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, there exists $(\mathsf{tr}', \phi') \in \mathsf{trace}(Q)$ such that $\mathsf{tr} = \mathsf{tr}'$ and $\phi \sim \phi'$. They are in *trace equivalence*, written $P \approx Q$, if $P \sqsubseteq Q$ and $Q \sqsubseteq P$.

The choice of $\mathcal{M}_\Sigma$ as well as the choice of public symbols allow to fine-tune what an attacker can observe. The set of public function symbols tell exactly which functions the attacker may use. Then the choice $\mathcal{M}_\Sigma$ defines when computations fail. For example, if $\mathcal{M}_\Sigma$ represents the set of terms with atomic keys only, then an attacker may potentially observe that some computation fails because he was able to inject a non atomic key.

Example 2.4.2. Let $n, k \in \mathcal{N}$ and consider the protocol $P = \mathsf{in}(c, x).\mathsf{out}(c, \mathsf{enc}(n, k))$ as well as the protocol $Q = \mathsf{in}(c, x).\mathsf{out}(c, \mathsf{enc}(\mathsf{enc}(n, x), k))$. An attacker may distinguish between P and Q by sending a non atomic data and observing whether the process can emit. Q will not be able to emit since its first encryption will fail. This attack would not have been detected if arbitrary terms were allowed in key position.

Example 2.4.3. The process P_{DS} presented in Example 2.2.1 models the Denning Sacco protocol. Assume now that we wish to check strong secrecy of the exchanged key, as received by the agent B. As discussed in Introduction, this can be expressed by checking whether $P^1_{\mathsf{DS}} \approx P^2_{\mathsf{DS}}$ where:

- P^1_{DS} is as P_{DS} but we add "$\beta_2 : \mathsf{out}(c_2, \mathsf{enc}(\mathsf{m}_1, y_{AB}))$" at the end of the process P_B;
- P^2_{DS} is as the protocol P_{DS} but we add at the end of P_B the instruction "$\mathsf{new}\, k.\beta_2 : \mathsf{out}(c_2, \mathsf{enc}(\mathsf{m}_2, k))$".

The terms m_1 and m_2 are two public constants from Σ_0.

While the key received by B cannot be learnt by an attacker, strong secrecy of this key is not guaranteed. Indeed, due to the lack of freshness, the same key can be sent several times to B, and this can be observed by an attacker. Formally, the attack is as follows. Consider the sequence

$$\mathsf{tr}' = \mathsf{tr}.\mathsf{out}(ch_2, \mathsf{w}_4).\mathsf{out}(c_B, ch_4).\mathsf{in}(ch_4, \mathsf{w}_3).\mathsf{out}(ch_4, \mathsf{w}_5)$$

where tr has been defined in Example 2.3.1. The attacker simply replays an old session. The resulting (unique) frames are

- $\phi'_1 = \phi \cup \{\mathsf{w}_4 \triangleright \mathsf{enc}(\mathsf{m}_1, k_{ab}),\ \mathsf{w}_5 \triangleright \mathsf{enc}(\mathsf{m}_1, k_{ab})\}$; and
- $\phi'_2 = \phi \cup \{\mathsf{w}_4 \triangleright \mathsf{enc}(\mathsf{m}_2, k),\ \mathsf{w}_5 \triangleright \mathsf{enc}(\mathsf{m}_2, k')\}$.

Then $(\mathsf{tr}', \phi'_1) \in \mathsf{trace}(P^1_{\mathsf{DS}})$ and $(\mathsf{tr}', \phi'_2) \in \mathsf{trace}(P^2_{\mathsf{DS}})$. However, we have that $\phi'_1 \not\sim \phi'_2$ since $\mathsf{w}_4 = \mathsf{w}_5$ in ϕ'_1 but not in ϕ'_2. Thus P^1_{DS} and P^2_{DS} are *not* in trace equivalence. To avoid this attack, the messages of the Denning-Sacco protocol shall include timestamps.

2.4.1 Determinate protocols

We consider *determinate* protocols as defined in [19], *i.e.*, we consider protocols in which the attacker knowledge is completely determined (up to static equivalence) by its past interaction with the protocol participants.

Definition 2.4.3. A protocol P is *determinate* if for any tr, and for any $(\mathcal{P}_1, \phi_1)$, $(\mathcal{P}_2, \phi_2)$ such that $P \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}_1, \phi_1)$, and $P \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}_2, \phi_2)$, we have that $\phi_1 \sim \phi_2$.

Determinate protocols are useful for the analysis of trace equivalence. Definition 2.4.2 can indeed be equivalently rewritten (see [19]) as:

Definition 2.4.4. Let P and Q be two determinate protocols. We have that $P \sqsubseteq Q$ if:

- for every $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and for every recipe R such that $R\phi\!\downarrow$ is a message, there exists $(\mathsf{tr}', \phi') \in \mathsf{trace}(Q)$ such that $\mathsf{tr} = \mathsf{tr}'$ and $R\phi'\!\downarrow$ is a message.
- for every $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and for every recipes R and R' such that $R\phi\!\downarrow$, $R'\phi\!\downarrow$ are messages and $R\phi\!\downarrow = R'\phi\!\downarrow$, there exists $(\mathsf{tr}', \phi') \in \mathsf{trace}(Q)$ such that $\mathsf{tr} = \mathsf{tr}'$ and $R\phi'\!\downarrow = R\phi'\!\downarrow$.

They are in *trace equivalence*, written $P \approx Q$, if $P \sqsubseteq Q$ and $Q \sqsubseteq P$.

This allows us to get rid of an alternation between quantifiers in the original definitions and makes possible the algorithms and proofs we describe in the next chapters.

2.4.2 Simple protocols

We then introduce the class of simple protocols, similar to the one introduced *e.g.* in [33].

Definition 2.4.5. A *simple protocol* P is a protocol of the form:

$$!\mathsf{new}\ c'_1.\mathsf{out}(c_1, c'_1).B_1 \mid \ldots \mid !\mathsf{new}\ c'_m.\mathsf{out}(c_m, c'_m).B_m \mid B_{m+1} \mid \ldots \mid B_{m+p}$$

where each B_i with $1 \leq i \leq m+p$ is a *basic process* on c_i, that is a ground process built using the following grammar:

$$B := 0 \mid \mathsf{in}(c_i, u).B \mid \mathsf{out}(c_i, u).B \mid \mathsf{let}\ x = v\ \mathsf{in}\ B \mid \mathsf{new}\, n.\, B$$

where $u \in \mathcal{T}(\Sigma_c, \mathcal{N} \cup \mathcal{X})$, $v \in \mathcal{T}(\Sigma, \mathcal{N} \cup \mathcal{X})$, and $n \in \mathcal{N}$. Moreover, we assume that $c_1, \ldots, c_m, c_{m+1}, \ldots, c_{m+p}$ are pairwise distinct.

Even if considering simple processes may seem to be restricted, in practice it is often the case that an attacker may identify processes through *e.g.* IP addresses and even sessions using sessions identifiers. Therefore, encoding protocols in such a class may be considered as a good practice since it allows to potentially discover more flaws. Indeed, it gives more power to the attacker and allows him to know from which agent he receives a message.

Lemma 2.4.1. A simple protocol is determinate.

Intuitively, as simple protocols use fresh channels on each of their branches, given a trace of that simple protocol, only one execution is possible, which then leads to a uniquely defined frame, up to renamings of nonces.

2.5 Type-compliant protocols

We describe several classes of protocols which will be used for our later results. They introduce the notion of typing system and type-compliance. These ideas are at the core of the results from Chapter 4 onward.

2.5.1 Typing systems

We now introduce the notion of *type-compliance* for protocols w.r.t. typing systems. In these protocols, terms are given a particular type and we require that types are preserved by unification and application of substitutions. These operations are indeed routinely used in decision procedures.

The type given to any term is defined by a *typing* system whose definition we give below.

Definition 2.5.1. A *typing system* is a pair $(\mathcal{T}, \delta)$ where $\mathcal{T}$ is a set of elements called *types*, and δ is a function mapping terms $t \in \mathcal{T}(\Sigma_c, \Sigma_0 \cup \mathcal{N} \cup \mathcal{X})$ to types τ in $\mathcal{T}$ such that:

- if t is a term of type τ and σ is a *well-typed* substitution, *i.e.* $\forall x \in dom(\sigma), \delta(x) = \delta(x\sigma)$, then $t\sigma$ is of type τ,
- for any terms t and t' with the same type, *i.e.* $\delta(t) = \delta(t')$ and which are unifiable, their most general unifier $(mgu(t, t'))$ is well-typed.

We further assume the existence of an infinite number of constants in Σ_0 (resp. variables in $\mathcal{X}$, names in $\mathcal{N}$) of any type.

A straightforward typing system is when all terms are of a unique type, say Msg. Of course, our typing result would then be useless to reduce the search space for attacks. Which typing system shall be used typically depends on the protocols under study.

Structure-preserving typing systems are example of useful typing systems, as it will be demonstrated in depth in Chapter 6. These more precise typing systems preserve the structure of terms and are defined as follows:

Definition 2.5.2. A *structure-preserving typing system* is a pair $(\mathcal{T}_0, \delta_0)$ where $\mathcal{T}_0$ is a set of elements called *atomic types*, and δ_0 is a function mapping atomic terms in $\Sigma_0 \cup \mathcal{N} \cup \mathcal{X}$ to types τ generated using the following grammar:

$$\begin{array}{rcll} \tau, \tau_1, \ldots, \tau_n & = & \tau_0 & \text{with } \tau_0 \in \mathcal{T}_0 \\ & & \mid\ \mathsf{f}(\tau_1, \ldots, \tau_n) & \text{with } \mathsf{f} \in \Sigma_{\mathsf{c}} \end{array}$$

We further assume the existence of an infinite number of constants in Σ_0 (resp. variables in $\mathcal{X}$, names in $\mathcal{N}$) of any type. Then, δ_0 is extended to constructor terms as follows:

$$\delta_0(\mathsf{f}(t_1, \ldots, t_n)) = \mathsf{f}(\delta_0(t_1), \ldots, \delta_0(t_n)) \text{ with } \mathsf{f} \in \Sigma_{\mathsf{c}}.$$

Note that a structure-preserving typing system is a special case of typing system, whose types are defined only slightly differently. Lemma 2.5.1 formally links the two definitions.

Lemma 2.5.1. If $(\mathcal{T}_0, \delta_0)$ is a structure-preserving typing system, $(img(\delta_0), \delta_0)$ is a typing system, where $img(\delta_0)$ is the image of the extension of δ_0 to arbitrary constructor terms.

Proof. Note that in Definition 2.5.1, $\mathcal{T}$ is the set of all types, whereas $\mathcal{T}_0$, as defined in Definition 2.5.2, only contains atomic types. The set of all types from $\mathcal{T}_0$ is obtained by looking at the image of the typing function extended to arbitrary types by induction on their structure: we set $\mathcal{T} = img(\delta_0)$ and $\delta = \delta_0$. Let us then verify that the two items of Definition 2.5.1 apply to $(\mathcal{T}, \delta)$:

- if t is a term, $\delta_P(t) = \tau$ and σ a well-typed substitution: let us reason by induction on t:
  - if t is a constant, name or variable such that $t \notin dom(\sigma)$, $t = t\sigma$, and then $\delta(t\sigma) = \tau$,
  - if t is a variable and $t \in dom(\sigma)$: as σ is well-typed, $\delta_P(x) = \delta_P(x\sigma)$ which amounts to $\delta_P(t) = \delta_P(t\sigma) = \tau$,
  - if $t = \mathsf{f}(t_1, \ldots, t_n)$ and for any i, $\delta_P(t_i) = \delta_P(t_i\sigma)$, then $\delta_P(\mathsf{f}(t_1, \ldots, t_n)) = \mathsf{f}(\delta_P(t_1), \ldots, \delta_P(t_n))$ by Definition 2.5.2, which is equal to, by induction hypothesis, $\mathsf{f}(\delta_P(t_1\sigma), \ldots, \delta_P(t_n\sigma))$ and the same as $\delta_P(\mathsf{f}(t_1\sigma, \ldots, t_n\sigma))$, by Definition 2.5.2 once again, leading to the result, $\delta_P(t) = \delta_P(t\sigma)$.

- for any terms t, t' such that $\delta_P(t) = \delta_P(t')$, $mgu(t, t')$ is well-typed: there again we prove the result by induction on the computation of the $mgu(t, t')$, as done *e.g.* in [6, Lemma 1].

□

2.5.2 Type compliance

The class of protocols we consider here is the class of protocols *type-compliant* w.r.t. some typing system. Their defining characteristic is that any two unifiable encrypted subterms are of the same type. The goal of this part is to state this hypothesis formally.

Due to the presence of replication, we need to consider two copies of protocols in order to consider different instances of the variables. Given a protocol P with replication, we define its 2-unfolding $\mathsf{unfold}^2(P)$ to be the protocol such that every occurrence of a process $!R$ in P is replaced by $R \mid R$, and some α-renaming is performed on one copy to ensure names and variables distinctness of the resulting process. Note that if P is a protocol that does not contain any replication, we have that $\mathsf{unfold}^2(P) = P$.

Example 2.5.1. Let $P_1 = \mathsf{in}(c, x).!\mathsf{new}\, k.\, \mathsf{in}(c, \mathsf{enc}(\langle x, y\rangle, k))$. We have that:
$\mathsf{unfold}^2(P_1) = \mathsf{in}(c, x).(\mathsf{new}\, k_1.\mathsf{in}(c, \mathsf{enc}(\langle x, y_1\rangle, k_1)) \mid \mathsf{new}\, k_2.\mathsf{in}(c, \mathsf{enc}(\langle x, y_2\rangle, k_2)))$

We write $St(t)$ for the set of *(syntactic) subterms* of a term t, and $ESt(t)$ the set of its *encrypted subterms*, *i.e.* $ESt(t) = \{u \in St(t) \mid u \text{ is of the form } \mathsf{enc}(u_1, u_2)\}$. We extend this notion to sets/sequences of terms, and to protocols as expected.

Definition 2.5.3. A protocol P is *type-compliant* w.r.t. a typing system $(\mathcal{T}, \delta)$ if for every $t, t' \in ESt(\mathsf{unfold}^2(P))$ we have that: t and t' unifiable implies that $\delta(t) = \delta(t')$.

Example 2.5.2. Going back to the protocols of Example 2.2.1 and 2.4.3, we consider the structure-preserving typing system generated from the set of atomic types

$$\mathcal{T}_{\mathsf{DS}} = \{\tau_{\mathsf{a}}, \tau_{\mathsf{b}}, \tau_{\mathsf{m}}, \tau_{kas}, \tau_{kbs}, \tau_{kab}\}$$

and the function δ_{DS} that associates the expected type to each constant/name, and the following type to variables:

- $\delta_{\mathsf{DS}}(x_{AB}) = \delta_{\mathsf{DS}}(y_{AB}) = \tau_{kab}$; and
- $\delta_{\mathsf{DS}}(x_B) = \mathsf{enc}(\langle \tau_{kab}, \tau_{\mathsf{a}}\rangle, \tau_{kbs})$.

Example 2.5.3. The protocol P^1_{DS} (resp. P^2_{DS}) is type-compliant w.r.t. the typing system given in Example 2.5.2. Indeed, the encrypted subterms of $\mathsf{unfold}^2(P^1_{\mathsf{DS}})$ are:

1. $t_A = \mathsf{enc}(\langle \mathsf{b}, x_{AB}, x_B\rangle, k_{as})$;
2. $t_{B1} = \mathsf{enc}(\langle y_{AB}, \mathsf{a}\rangle, k_{bs})$;
3. $t_{B2} = \mathsf{enc}(\mathsf{m}_1, y_{AB})$;
4. $t_{S1} = \mathsf{enc}(\langle \mathsf{b}, k_{ab}, \mathsf{enc}(\langle k_{ab}, \mathsf{a}\rangle, k_{bs})\rangle, k_{as})$; and
5. $t_{S2} = \mathsf{enc}(\langle k_{ab}, \mathsf{a}\rangle, k_{bs})$

as well as the renaming of these terms obtained by replacing k_{ab}, x_{AB}, y_{AB}, and x_B with fresh names/variables of the same type.

It is easy to check that the type-compliance condition is satisfied for any pair of terms. For instance, we have that t_A and t_{S1} are unifiable, and they have indeed the same type:

$$\begin{aligned} \delta_{\mathsf{DS}}(t_A) &= \mathsf{enc}(\langle \tau_{\mathsf{b}}, \tau_{kab}, \mathsf{enc}(\langle \tau_{kab}, \tau_{\mathsf{a}}\rangle, \tau_{kbs})\rangle, \tau_{kas}) \\ &= \delta_{\mathsf{DS}}(t_{S1}). \end{aligned}$$

As shown in the following example, not all protocols are type-compliant w.r.t. a structure-preserving typing system.

Example 2.5.4. For instance, consider the following protocol:

$$P \quad = \quad !\,\mathsf{new}\, c.\mathsf{out}(c, c').\ \mathsf{in}(c', \mathsf{enc}(x, k)).\ \mathsf{out}(c', \mathsf{enc}(\langle x, x\rangle, k)$$

We have that $t_1 = \mathsf{enc}(x, k)$ and $t_2 = \mathsf{enc}(\langle x, x\rangle, k)$ are both in $ESt(\mathsf{unfold}^2(P))$ as well as the terms t_1' and t_2' obtained from t_1 and t_2 by simply renaming x with another variable, say x', having the same type as x. The two terms t_1 and t_2' are unifiable, and thus should receive the same type, but this would imply that $\delta(x) = \langle \delta(x'), \delta(x')\rangle$, and thus x can not receive the same type as x'.

2.5.3 Tagged protocols

Finally, as an instance of our definitions, we consider tagged protocols, for a notion of tagging similar to one introduced by Blanchet and Podelski [15]. They form an interesting particular case of type-compliant protocols. Assume given a protocol P and an unfolding P' of it (remember that when computing $\mathsf{unfold}^2(P)$ names and variables are renamed to avoid clashes). Let u be a term in $\mathcal{T}(\Sigma_c, \Sigma_P \cup \mathcal{N}_P' \cup \mathcal{X}_P')$ where Σ_P, $\mathcal{N}_P'$, $\mathcal{X}_P'$ are the constants, names, and variables occurring in P', we denote by $\overline{u}$ the transformation that replaces any name and variable occurring in u by its antecedent in $\mathcal{N}_P$ and $\mathcal{X}_P$ where $\mathcal{N}_P$ and $\mathcal{X}_P$ are the names and variables occurring in P.

Definition 2.5.4. A protocol P is *tagged* if there exists a substitution σ_P such that for any $s_1, s_2 \in ESt(\mathsf{unfold}^2(P))$ with s_1 and s_2 unifiable, we have that $\overline{s_1\sigma_P} = \overline{s_2\sigma_P}$.

The protocols introduced in Example 2.4.3 are example of tagged protocols. If a protocol is not naturally tagged, tagging can easily be enforced by labelling encrypted terms, as proposed in [15].

Definition 2.5.5. A protocol P is *strongly tagged* if:

1. any term in $ESt(P)$ is of the form $\mathsf{enc}(\langle \mathsf{c}, m\rangle, k)$ for some $\mathsf{c} \in \Sigma_0$; and
2. there exists σ_P such that for any $s, t \in ESt(P)$ with $s = \mathsf{enc}(\langle \mathsf{c}_0, s_1\rangle, s_2)$ and $t = \mathsf{enc}(\langle \mathsf{c}_0, t_1\rangle, t_2)$ for some $\mathsf{c}_0 \in \Sigma_0$, we have that $s\sigma_P = t\sigma_P$.

The second condition requires that there is a substitution that unifies any two tagged terms unless their tags differ. This condition is easy to achieve for executable protocols. More precisely, assume a protocol admits an execution where each protocol step (in and out) is executed once (*i.e.* there is one honest execution). This protocol can be easily strongly tagged by adding a distinct tag in each encrypted term.

Lemma 2.5.2. Let P be a protocol. If P is strongly tagged then P is tagged.

Proof. Let us assume P is a strongly tagged protocol and σ_P be the substitution as in Definition 2.5.5. Let $s_1, s_2 \in ESt(\mathsf{unfold}^2(P))$ such that there exists σ with $s_1\sigma = s_2\sigma$. As both terms are of the form $\mathsf{enc}(\langle \mathsf{c}, u_i\rangle, v_i)$ for some u_i and v_i and are unifiable, they share the same tagging constant c. Then $s_1\sigma_P = s_2\sigma_P$ by Definition 2.5.5, and thus, *a fortiori*, $\overline{s_1\sigma_1} = \overline{s_2\sigma_2}$. □

Example 2.5.5. The Otway-Rees protocol [30] is a key distribution protocol using symmetric encryption and a trusted server. It can be described informally as follows:

1. $A \to B:\ M, A, B, \{N_a, M, A, B\}_{K_{as}}$
2. $B \to S:\ M, A, B, \{N_a, M, A, B\}_{K_{as}}, \{N_b, M, A, B\}_{K_{bs}}$
3. $S \to B:\ M, \{N_a, K_{ab}\}_{K_{as}}, \{N_b, K_{ab}\}_{K_{bs}}$
4. $B \to A:\ M, \{N_a, K_{ab}\}_{K_{as}}$

where $\{m\}_k$ denotes the symmetric encryption of a message m with key k, A and B are agents trying to authenticate each other, S is a trusted server, K_{as} (resp. K_{bs}) is a long term key shared between A and S (resp. B and S), N_a and N_b are nonces generated by A and B, K_{ab} is a session key generated by S, and M is a session identifier.

We propose a modelling of the Otway-Rees protocol in our formalism. We use restricted channels to model the use of unique session identifiers used along an execution of the protocol. Below, k_{as}, k_{bs}, m, n_a, n_b, k_{ab} are names, whereas a and b are constants from Σ_0. We denote by $\langle x_1, \ldots, x_{n-1}, x_n\rangle$ the term $\langle x_1, \langle \ldots \langle x_{n-1}, x_n\rangle\rangle\rangle$.

$$P_{\mathsf{OR}} = !\,\mathsf{new}\ c_1.\mathsf{out}(c_A, c_1).P_A \mid !\,\mathsf{new}\ c_2.\mathsf{out}(c_B, c_2).P_B \mid !\,\mathsf{new}\ c_3.\mathsf{out}(c_S, c_3).P_S$$

where the processes P_A, P_B are given below, and P_S can be defined in a similar way.

$$\begin{aligned} P_A = \mathsf{new}\ m.\mathsf{new}\ n_a.\,&\mathsf{out}(c_1, \langle m, \mathsf{a}, \mathsf{b}, \mathsf{enc}(\langle n_a, m, \mathsf{a}, \mathsf{b}\rangle, k_{as})\rangle).\\ &\mathsf{in}(c_1, \langle m, \mathsf{enc}(\langle n_a, x_{ab}\rangle, k_{as})\rangle);\end{aligned}$$

$$\begin{aligned} P_B = \mathsf{in}(c_2, \langle y_m, \mathsf{a}, \mathsf{b}, y_{as}\rangle).\mathsf{new}\ n_b.&\mathsf{out}(c_2, \langle y_m, \mathsf{a}, \mathsf{b}, y_{as}, \mathsf{enc}(\langle n_b, y_m, \mathsf{a}, \mathsf{b}\rangle, k_{bs})\rangle).\\ &\mathsf{in}(c_2, \langle y_m, z_{as}, \mathsf{enc}(\langle n_b, y_{ab}\rangle, k_{bs})\rangle).\mathsf{out}(c_2, \langle y_m, z_{as}\rangle)\end{aligned}$$

In our modelling, P_{OR} is not tagged. For instance, let us consider the two terms $s_1 = \mathsf{enc}(\langle n_a, m, \mathsf{a}, \mathsf{b}\rangle, k_{as})$ and $s_2 = \mathsf{enc}(\langle n_a, x_{ab}\rangle, k_{as})$. Both are encrypted subterms of P_A (and thus of $\mathsf{unfold}^2(P_{\mathsf{OR}})$) and s_1 and s_2 are unifiable. Now, consider $s_3 = \mathsf{enc}(\langle z_a, k_{ab}\rangle, k_{as})$. Actually, s_3 is an encrypted subterm of P_S which is unifiable with s_2. However, there exists no substitution σ such that $\overline{s_1\sigma} = \overline{s_2\sigma} = \overline{s_3\sigma}$.

We can consider a strongly tagged, and safer, version of the Otway-Rees protocol by introducing 4 different tags, denoted 1,2,3 and 4, that are modelled using constants from Σ_0.

$$P'_{\mathsf{OR}} = !\,\mathsf{new}\ c_1.\mathsf{out}(c_A, c_1).P'_A \mid !\,\mathsf{new}\ c_2.\mathsf{out}(c_B, c_2).P'_B \mid !\,\mathsf{new}\ c_3.\mathsf{out}(c_S, c_3).P'_S$$

$$\begin{aligned} P'_A = \quad &\mathsf{new}\ m.\mathsf{new}\ n_a.\,\mathsf{out}(c_1, \langle m, \mathsf{a}, \mathsf{b}, \mathsf{enc}(\langle 1, n_a, m, \mathsf{a}, \mathsf{b}\rangle, k_{as})\rangle).\\ &\mathsf{in}(c_1, \langle m, \mathsf{enc}(\langle 2, n_a, x_{ab}\rangle, k_{as})\rangle)\\ P'_B = \quad &\mathsf{in}(c_2, \langle y_m, \mathsf{a}, \mathsf{b}, y_{as}\rangle).\\ &\mathsf{new}\ n_b.\,\mathsf{out}(c_2, \langle y_m, \mathsf{a}, \mathsf{b}, y_{as}, \mathsf{enc}(\langle 3, n_b, y_m, \mathsf{a}, \mathsf{b}\rangle, k_{bs})\rangle).\\ &\mathsf{in}(c_2, \langle y_m, z_{as}, \mathsf{enc}(\langle 4, n_b, y_{ab}\rangle, k_{bs})\rangle).\mathsf{out}(c_2, \langle y_m, z_{as}\rangle)\\ P'_S = \quad &\mathsf{in}(c_3, \langle z_m, \mathsf{a}, \mathsf{b}, \mathsf{enc}(\langle 1, z_a, z_m, \mathsf{a}, \mathsf{b}\rangle, k_{as}), \mathsf{enc}(\langle 3, z_b, z_m, \mathsf{a}, \mathsf{b}\rangle, k_{bs})\rangle).\\ &\mathsf{new}\ k_{ab}.\,\mathsf{out}(c_3, \langle z_m, \mathsf{enc}(\langle 2, z_a, k_{ab}\rangle, k_{as}), \mathsf{enc}(\langle 4, z_b, k_{ab}\rangle, k_{bs})\rangle)\end{aligned}$$

We can show that P'_{OR} is strongly tagged: consider the natural execution of P'_{OR}, matching inputs and outputs as intended. From this execution we can define:

$$\begin{aligned}\sigma_P = \{&x_{ab} \rhd k_{ab},\ y_m \rhd m,\ y_{as} \rhd \mathsf{enc}(\langle 1, n_a, m, \mathsf{a}, \mathsf{b}\rangle, k_{as})\rangle,\\ &z_{as} \rhd \mathsf{enc}(\langle 2, n_a, k_{ab}\rangle, k_{as}),\ z_m \rhd m,\ z_a \rhd n_a,\ z_b \rhd n_b\}.\end{aligned}$$

It is then easy to check that for any two terms s_1 and s_2 that are unifiable, their instances by σ_P are actually identical.

For any tagged protocol, we can infer a finite typing system, and show the type-compliance of the tagged protocol w.r.t. this typing system.

Definition 2.5.6. Let P be a tagged protocol, and σ_P the substitution witnessing this fact. Let Σ_P, $\mathcal{N}_P$, $\mathcal{X}_P$ be respectively the constants, names, and variables occurring in P. We consider the function δ_P, inductively defined on $\mathcal{T}(\Sigma_{\mathsf{c}}, \Sigma_P \cup \mathcal{N}_P \cup \mathcal{X}_P)$ as follows:

- $\delta_P(x) = \overline{x\sigma_P}$ for any variable that occurs in P;
- $\delta_P(a) = a$ for any name, constant that occurs in P.
- $\delta(\mathsf{f}(t_1, t_2)) = \mathsf{f}(\delta(t_1), \delta(t_2))$ for $\mathsf{f} \in \{\mathsf{enc},\ \langle\ \rangle\}$.

The image of δ_P is a set of types, denoted $\mathcal{T}_p$. The function δ_P is then extended arbitrarily to the remaining names, variables, constants such that there is an infinite set of names, variables, constants of each type in $\mathcal{T}_p$. This extends δ_P on $\mathcal{T}(\Sigma_{\mathsf{c}}, \Sigma_0 \cup \mathcal{N} \cup \mathcal{X})$ using the recursive definition: $\delta(\mathsf{f}(t_1, t_2)) = \mathsf{f}(\delta(t_1), \delta(t_2))$ when $\mathsf{f} \in \{\mathsf{enc},\ \langle\ \rangle\}$.

Any tagged protocol is actually type-compliant w.r.t. its induced typing system.

Proposition 2.5.1. Let P be a tagged protocol and let $(\mathcal{T}_P, \delta_P)$ as defined in Definition 2.5.6.

1. $(\mathcal{T}_P, \delta_P)$ is a typing system. We say that it is the typing system *induced by* P.
2. P is type-compliant w.r.t. $(\mathcal{T}_P, \delta_P)$.

Proof. First we prove by induction on terms t and t' in Definition 2.5.1 that (T_P, δ_P) as introduced in Definition 2.5.6 is a typing system. Then we prove that if P is tagged then P is type-compliant w.r.t. (T_P, δ_P). Indeed, let $s, t \in ESt(\mathsf{unfold}^2(P))$ and a substitution σ such that $s\sigma = t\sigma$. We need to prove that $\delta_P(s) = \delta_P(t)$. Because P is tagged, $\overline{s\sigma_P} = \overline{t\sigma_P}$. Moreover, $\delta_P(s) = \delta_P(\overline{s\sigma_P})$ and $\delta_P(t) = \delta_1(\overline{t\sigma_P})$. Hence $\delta_P(s) = \delta_P(\overline{s\sigma_P}) = \delta_P(\overline{t\sigma_P}) = \delta_P(t)$. □

2.6 Conclusion

This chapter presented the general model needed to efficiently state the results from the following chapters. It also introduces the general classes of protocols, *determinate*, *simple* and *type-compliant* protocols, which are needed hypotheses to prove our main results. Determinate and simple protocols are both incarnations of the intuitive need for "deterministic" protocols explained in introduction as they limit, up to some equivalence, the number of executions for a given trace and thus constrain the branching nature of the search space when hoping to check trace equivalence. Type-compliant protocols on the other hand are related to tagging schemes, of which they form a generalisation and will allow us in Chapter 4 to state our typing result to greatly reduce the said search space.

Chapter 3

How to get rid of nonces

When proving security properties, it is important to obtain guarantees for an unlimited number of sessions. Unfortunately, it is well known that even secrecy is undecidable [44] in this context. Therefore, a standard restriction consists in bounding the number of nonces (and keys). Under this assumption, several decidability results have been established for secrecy [44, 15, 31], and other will be for trace equivalence in Chapters 5 and 7.

Replacing nonces by constants is sound in the context of secrecy properties. More precisely, assuming that $\overline{P}$ is obtained from the security protocol P by replacing nonces (and keys) by constants, whenever $\overline{P}$ is secure (w.r.t. a trace property such as secrecy) then P is secure as well. Indeed, replacing nonces by constants may only introduce more attacks, since it may only create more equalities, as long as the protocol P under study does not have else branches. Therefore, the decidability results developed for secrecy (*e.g.* [44, 15, 31]) may be seen as proof techniques: if $\overline{P}$ falls in a decidable class and can be shown to be secure then the protocol P is secure as well. Unfortunately, such an approach is no longer valid in the context of equivalence properties. Indeed, consider the processes:

$$P = !\ \mathsf{new}\ n.\mathsf{out}(c, \{n\}_k) \ \text{ and } \ Q = !\ \mathsf{out}(c, \{n\}_k).$$

Intuitively, both processes send out an arbitrary number of messages on the public channel c. The process P sends out each time a fresh nonce n encrypted by a (secret) key k while Q always sends the same message. We assume here that encryption is not randomised. Clearly, the processes P and Q are not in equivalence (denoted $P \not\approx Q$) since an attacker can easily notice that P sends distinct messages while Q sends identical messages. However, abstracting away fresh names with constants, the resulting equivalence holds (denoted $\overline{P} \approx \overline{Q}$). Indeed, the two resulting processes are actually identical: $\overline{P} = \overline{Q} = !\ \mathsf{out}(c, \{n\}_k)$. This illustrates that $\overline{P} \approx \overline{Q} \not\Rightarrow P \approx Q$.

In this chapter, we identify a technique to (soundly) get rid of freshly generated data (*e.g.* nonces, keys). The main idea consists in introducing an additional copy of each replicated nonce. More precisely, we show that:

$$!\,\overline{P} \mid P^\star \approx !\,\overline{Q} \mid Q^\star \quad \Rightarrow \quad !P \approx !Q$$

where $P^\star$ is obtained from P by renaming all fresh nonces and keys to distinct (fresh) constants. Our result holds for simple processes, a notion described in Section 2.4.2. We consider a large family of primitives, provided that they can be described by a destructor/constructor theory with no critical pair. In particular, our technique allows one to deal with standard primitives (asymmetric and symmetric encryption, hash, signatures, MACs) as well as *e.g.* blind signatures and zero-knowledge proofs. In Chapters 5 and 7 we will present decidability results developed for protocols without nonces which could be applied to study the security of protocols *with* nonces thanks to this result.

Related work. Abstracting nonces and keys by constants is known to be sound for secrecy properties as part of the "folklore". We did not find a precise reference for this result. A related result is a reduction to two agents [32] for trace properties. Reducing the number of nonces can be obtained in a similar way.

The tool ProVerif [13, 14] also makes use of an abstraction for fresh data. In case of secrecy, nonces are abstracted by functions applied to the process inputs. In case of equivalence properties, nonces are additionally given a counter

(termination is of course not guaranteed). The abstraction technique is therefore more precise than using only constants but seems dedicated to the internal behaviour of the ProVerif tool.

3.1 Our hypotheses

Our technique soundly abstracts nonces and keys for trace equivalence, for *simple protocols* and for a large family of security primitives, namely *adequate theories*, that we define in this section.

In order to establish our result, we have to ensure that considering two distinct constants instead of fresh nonces is sufficient. We need this property to hold on terms first. Intuitively, when a term cannot be reduced further, it should be possible to isolate two nonces that cause the reduction to fail. This is indeed the case for a large class of primitives. We formalise this notion as follows:

Definition 3.1.1. Given a signature $\Sigma = \Sigma_c \uplus \Sigma_d$, a convergent rewriting system $\mathcal{R}$, and a set of messages $\mathcal{M}_\Sigma$, we say that the theory $(\Sigma, \mathcal{R})$ is *adequate* w.r.t. $\mathcal{M}_\Sigma$ when for any term $t \in \mathcal{T}(\Sigma, \mathcal{N}) \setminus \mathcal{M}_\Sigma$ in normal form, there exist $n_1, n_2 \in \mathcal{N}$ such that for any renaming ρ with $\rho(n_1) \neq \rho(n_2)$ then $t\rho\downarrow \notin \mathcal{M}_\Sigma$.

Intuitively, we require that whenever a term t is not a message, it is possible to fix two names of t such that any renaming of t (preserving these two names) is still not a message. We could generalise our criterion to n-adequate theories where the number of names that need to fixed is bounded by n but two names are actually sufficient to deal with most of the theories.

Example 3.1.1. The theory described in Example 2.1.1 is adequate w.r.t. to the two notions of messages $\mathcal{M}_c$ and $\mathcal{M}_{\mathsf{atomic}}$ that have been introduced. Intuitively, when a term is not a message, either this property is actually stable for any renaming (*e.g.* $\mathsf{sdec}(n, k)$) or is due to the failure of a decryption (*e.g.* $\mathsf{sdec}(\mathsf{senc}(n, k), k')$). In such a case, maintaining the disequality between the terms modelling the encryption and the decryption keys is sufficient to ensure that the resulting term will not become a message.

Since the adequacy hypothesis might be cumbersome to prove by hand for each theory, we exhibit a simple criterion that ensures adequacy: the absence of critical pair.

Definition 3.1.2. Given a signature $\Sigma = \Sigma_c \uplus \Sigma_d$, and a rewriting system $\mathcal{R}$, we say that the theory $(\Sigma, \mathcal{R})$ has *no critical pair* if ℓ_1 and ℓ_2 are not unifiable for any distinct rules $\ell_1 \to r_1$, and $\ell_2 \to r_2$ in $\mathcal{R}$.

Our notion of critical pairs actually coincide with the usual one for the theories we consider. Indeed, rewrite rules are all of the form $\ell \to r$ such that the head symbol of ℓ is a destructor symbol and destructors may not appear anywhere else in ℓ nor r. Theories without critical pairs are convergent and adequate.

Lemma 3.1.1. Given a signature $\Sigma = \Sigma_c \uplus \Sigma_d$, a rewriting system $\mathcal{R}$, and a set of messages $\mathcal{M}_\Sigma$ such that $\mathcal{T}(\Sigma_c, \mathcal{N}) \setminus \mathcal{M}_\Sigma$ is stable by renaming. If the theory $(\Sigma, \mathcal{R})$ has no critical pair, then $(\Sigma, \mathcal{R})$ is convergent and adequate w.r.t. $\mathcal{M}_\Sigma$.

Proof. To prove the convergence of $\mathcal{R}$, we need to show $\mathcal{R}$ is both terminating and confluent. Because each rewrite rule strictly decreases the number of destructors in a term, the termination is trivial. Confluence then stems from local confluence (Newman's lemma), which comes from the fact that, at any given position in a term, only one rule of $\mathcal{R}$ can be applied (as $\mathcal{R}$ has no critical pair) and reductions have to follow an innermost strategy (because rules can only be applied on arguments without destructors).

Let $t \in \mathcal{T}(\Sigma, \mathcal{N}) \setminus \mathcal{M}_\Sigma$ be a term in normal form. We want to prove there exist $n_1, n_2 \in \mathcal{N}$ such that for any renaming ρ with $\rho(n_1) \neq \rho(n_2)$, $t\rho\downarrow \notin \mathcal{M}_\Sigma$. If $t \in \mathcal{T}(\Sigma_c, \mathcal{N}) \setminus \mathcal{M}_\Sigma$, then the result directly holds as $\mathcal{T}(\Sigma_c, \mathcal{N}) \setminus \mathcal{M}_\Sigma$ is stable by renaming. Now, t must then contain at least one destructor. Let p be one of the lowest positions such that $t = C[\mathsf{g}(t_1, \ldots, t_k)]_p$ where $\mathsf{g} \in \Sigma_d$ and for any position $q > p$ (we note $q > p$ when p is a prefix of q), the symbol at position q in t is a constructor. In particular, $t_1, \ldots, t_k \in \mathcal{T}(\Sigma_c, \mathcal{N})$.

We consider the case where there exists a renaming ρ such that $\mathsf{g}(t_1,\dots,t_k)\rho \to s$ using some *linear* rule $R \in \mathcal{R}$. Because rules in $\mathcal{R}$ start with a destructor, this rewrite rule can only be applied at position ϵ in $\mathsf{g}(t_1,\dots,t_k)\rho$. Because R is linear, we also have that $\mathsf{g}(t_1,\dots,t_k) \to s'$ using the same rule R and $s = s'\rho$, and thus t would not be in normal form. Hence we can assume that for any renaming ρ, no linear rule can be applied on $\mathsf{g}(t_1,\dots,t_n)\rho$.

Let $n_0 \in \mathcal{N}$ and ρ_{n_0} the renaming such that $\rho_{n_0}(n) = n_0$ for any $n \in \mathcal{N}$. Let ρ_n (resp. ρ_m) be a renaming such that there exists $n_1, n_2 \in \mathcal{N}$ (resp. $m_1, m_2 \in \mathcal{N}$) such that $\rho_n(n_1) \neq \rho_n(n_2)$ (resp. $\rho_m(m_1) \neq \rho_m(m_2)$). Let us further assume there exists some *non-linear* rule $R_n \in \mathcal{R}$ such that $\mathsf{g}(t_1,\dots,t_k)\rho_n \to s_n$ (resp. some *non-linear* rule $R_m \in \mathcal{R}$ such that $\mathsf{g}(t_1,\dots,t_k)\rho_m \to s_m$). We want to prove first that $R_n = R_m$. By definition of ρ_{n_0} and ρ_n (resp. ρ_m), there exists a renaming δ_n (resp. δ_m) such that $\rho_n\delta_n = \rho_{n_0}$ (resp. $\rho_m\delta_m = \rho_{n_0}$) and $\mathsf{g}(t_1,\dots,t_k)\rho_n\delta_n \to s_n\delta_n$ using R_n (resp. $\mathsf{g}(t_1,\dots,t_k)\rho_m\delta_m \to_{R_m} s_m\delta_m$ using R_m). So we get that $\mathsf{g}(t_1,\dots,t_k)\rho_{n_0} \to s_n\delta_n$ using R_n and $\mathsf{g}(t_1,\dots,t_k)\rho_{n_0} \to s_m\delta_m$ using R_m. Thus, this means that the left-hand side of R_n and R_m are unifiable. Because $\mathcal{R}$ contains no critical pair, we necessarily have that $R_n = R_m$. Thus there exist exactly one (non-linear) rule R which can reduce $\mathsf{g}(t_1,\dots,t_k)\rho_n$ for any such ρ_n. Let $R = \mathsf{g}(u_1,\dots,u_k) \to r$ that rule. As $\mathsf{g}(t_1,\dots,t_k)$ is in normal form (and thus R cannot be applied) while $\mathsf{g}(t_1,\dots,t_k)\rho_n \to s_n$ using R for every ρ_n, we know that there exist two positions p_1 and p_2, two indices i and j, and a variable x such that $u_i|_{p_1} = u_j|_{p_2} = x$. Actually, we can even assume there exist two leaf positions $q_1 > p_1$ and $q_2 > p_2$ such that $t_i|_{q_1} \neq t_j|_{q_2}$ but $t_i\rho_n|_{q_1} = t_j\rho_n|_{q_2}$. Note that t_i and $t_i\rho_n$ (resp. t_j and $t_j\rho_n$) share the same leaf positions. Necessarily $t_i|_{q_1}, t_j|_{q_2} \in \mathcal{N}$, as otherwise $t_i\rho_n|_{q_1} \neq t_j\rho_n|_{q_2}$. We have that $t_i|_{q_1} \neq t_j|_{q_2}$ and both are names. Let ρ_r be a renaming such that $\rho_r(t_i|_{q_1}) \neq \rho_r(t_j|_{q_2})$. We have that $t_i|_{q_1}\rho_r = t_i|_{q_1}$ and $t_i|_{q_1}\rho_r \neq t_j|_{q_2}\rho_r$. The latter implies that $t_i|_{p_1}\rho_r \neq t_j|_{p_2}\rho_r$, and as such $\mathsf{g}(t_1,\dots,t_k)\rho_r$ cannot reduce with rule R, and consequently cannot reduce with any rule of $\mathcal{R}$. Thereby, $\mathsf{g}(t_1,\dots,t_k)\!\downarrow = \mathsf{g}(t_1,\dots,t_k) \notin \mathcal{T}(\Sigma_c,\mathcal{N})$.

Finally $\mathsf{g}(t_1,\dots,t_k)$ is in normal form and not a message, which ensures for any position $q < p$, $t|_q \notin \mathcal{M}_\Sigma$ (because t_q would contain the destructor g) and $t|_q$ cannot reduce with any rule of $\mathcal{R}$, as reduction requires that any variable must be instantiated with a message and one of its subterm is not a message (the rules themselves contain exactly one destructor in top-level position). □

This lemma allows us to conclude that many theories used in practice to model security protocols are actually adequate. This is the case of the theory given in Example 2.1.1, and the theories that are presented below.

Standard cryptographic primitives. We may enrich the theory described in Example 2.1.1 with function symbols to model asymmetric encryption, and digital signatures.

$$\Sigma^+ = \Sigma \cup \{\mathsf{aenc}, \mathsf{adec}, \mathsf{sign}, \mathsf{checksign}, \mathsf{getmsg}, \mathsf{pub}, \mathsf{priv}, \mathsf{ok}\}.$$

Symbols adec/aenc and sign/checksign of arity 2 are used to model asymmetric encryption and signature, whereas pub/priv of arity 1 will be used to model key pairs, and the symbol priv will be part of the signature Σ_{priv}. The symbol getmsg may be used in case we want to consider a signature algorithm that does not protect the signed message. The corresponding rewrite rules are defined as follows:

$$\begin{array}{rcl} \mathsf{checksign}(\mathsf{sign}(x,\mathsf{priv}(y)),\mathsf{pub}(y)) & \to & \mathsf{ok} \\ \mathsf{getmsg}(\mathsf{sign}(x,\mathsf{priv}(y)) & \to & x \\ \mathsf{adec}(\mathsf{aenc}(x,\mathsf{pub}(y)),\mathsf{priv}(y)) & \to & x \end{array}$$

Regarding the notion of messages, a reasonable choice for $\mathcal{M}_{\Sigma^+}$ is to consider $\mathcal{M}_c^+ = \mathcal{T}(\Sigma_c \uplus \{\mathsf{aenc}, \mathsf{sign}, \mathsf{pub}, \mathsf{priv}, \mathsf{ok}\}, \mathcal{N})$ the set of all ground constructor terms. We may also restrict $\mathcal{M}_{\Sigma_+}$ in various ways to only allow some specific terms in key positions.

Blind signatures. The following theory is often used to model blind signatures (see *e.g.* [39]), checksign and unblind are the only destructor symbols.

$$\begin{array}{rcl} \mathsf{checksign}(\mathsf{sign}(x,\mathsf{priv}(y)),\mathsf{pub}(y)) & \to & x \\ \mathsf{unblind}(\mathsf{blind}(x,y),y) & \to & x \\ \mathsf{unblind}(\mathsf{sign}(\mathsf{blind}(x,y),\mathsf{priv}(z)),y) & \to & \mathsf{sign}(x,\mathsf{priv}(z)) \end{array}$$

Zero-knowledge proofs. A typical signature for representing zero-knowledge proofs is $\Sigma_{\mathsf{ZKP}} = \{\mathsf{Verify}, \mathsf{ZKP}, \mathsf{ok}\}$ where ZKP represents a zero-knowledge proof and Verify models the verification of the proof. To ease the presentation, we present how to model the proof of a particular statement, namely the fact that a ciphertext is the encryption of either 0 or 1. Such proofs are thoroughly used for example in the context of e-voting protocols such as Helios. In particular, the theory we consider here has been introduced in [37]. Specifically, let $\Sigma^{+}_{\mathsf{ZKP}} = \Sigma_{\mathsf{ZKP}} \uplus \{\mathsf{raenc}, \mathsf{radec}, \mathsf{pub}, \mathsf{priv}, 0, 1\}$ and consider the following rewrite rules.

$$\begin{array}{rcl} \mathsf{radec}(\mathsf{raenc}(x, z, \mathsf{pub}(y)), \mathsf{priv}(y)) & \to & x \\ \mathsf{Verify}(\mathsf{ZKP}(x, \mathsf{raenc}(0, x, \mathsf{pub}(y)), \mathsf{pub}(y)), \mathsf{raenc}(0, x, \mathsf{pub}(y)), \mathsf{pub}(y)) & \to & \mathsf{ok} \\ \mathsf{Verify}(\mathsf{ZKP}(x, \mathsf{raenc}(1, x, \mathsf{pub}(y)), \mathsf{pub}(y)), \mathsf{raenc}(1, x, \mathsf{pub}(y)), \mathsf{pub}(y)) & \to & \mathsf{ok} \end{array}$$

The symbol raenc represents randomised asymmetric encryption as reflected by the first rewrite rule. The two last rules ensure that a proof is valid only if the corresponding ciphertext contains either 0 or 1 and nothing else. Many variants of zero-knowledge proofs can be modelled in a very similar way.

3.2 Getting rid of nonces

As explained in introduction, our main contribution is to provide a transformation that soundly abstracts nonces. Informally, we prove an implication of the following form:

$$!\,\overline{P} \mid P^{\star} \;\approx\; !\,\overline{Q} \mid Q^{\star} \quad \Rightarrow \quad !P \approx !Q$$

where $\overline{P}$ is obtained from P by replacing nonces by constants, and $P^{\star}$ is a copy of P. Before defining formally this transformation in Section 3.2.1, we introduce in Section 3.1 which hypotheses are required for the soundness of our transformation.

3.2.1 Our transformation

We now explain how to formally get rid of nonces. Our transformation is actually modular w.r.t. which nonces shall be abstracted. Let P be a simple process in which any name is bound at most once. This means that any name that does not occur explicitly in the scope of a restriction is distinct from those introduced by the new operator. Moreover, a same name can not be introduced twice by the operator new. Our transformation is parametrised by a set of names N which correspond to the new instructions that we want to remove (typically those under a replication).

We denote by $\overline{P}^{\mathsf{N}}$ (or simply $\overline{P}$ when N is clear from the context) the process obtained from P by removing every instruction $\mathsf{new}\ n$ for any $n \in \mathsf{N}$. Given $B(c)$ a basic process built on channel c, we denote by $B^{\star}(c^{\star})$ the process obtained from B by applying a bijective alpha-renaming on each name bound by a new instruction and replacing each occurrence of the channel c with the channel $c^{\star}$ (that is assumed to be fresh).

Example 3.2.1. Consider the process $P = !\mathsf{new}\ c'.\mathsf{out}(c, c').B$ where B is a basic process built on channel c'. Let $B = \mathsf{new}\ n.\mathsf{out}(c', \mathsf{senc}(n, k))$, and $\mathsf{N} = \{n\}$. We have that:

1. $\overline{P} = !\mathsf{new}\ c'.\mathsf{out}(c, c').\mathsf{out}(c', \mathsf{senc}(n, k))$, and
2. $B^{\star}(c^{\star}) = \mathsf{new}\, n^{\star}.\mathsf{out}(c^{\star}, \mathsf{senc}(n^{\star}, k))$.

Note that B and $B^{\star}(c^{\star})$ are identical up to the fact that they proceed on different channel. The transformation $\star$ applied on the basic process is just here to emphasise the fact that bound names are renamed to avoid some confusion due to name clashes.

Now, our transformation consists of combining these two building blocks. When removing fresh names from a process P, we keep a copy of one of the replicated basic processes of P, identified by its channel c. More formally, given a simple process P of the form $P = !\,\mathsf{new}\, c'.\mathsf{out}(c, c').B \mid P'$, and a set of names N, the resulting process $\overline{P}^{\mathsf{N},c}$ is defined as follows:

$$\overline{P}^{\mathsf{N},c} \stackrel{\mathsf{def}}{=} \overline{P}^{\mathsf{N}} \mid B^\star(c^\star).$$

Sometimes we simply write $\overline{P}^c$ instead of $\overline{P}^{\mathsf{N},c}$ when N is clear from the context.

Example 3.2.2. Continuing Example 3.2.1, we have that:

$$\overline{P}^{\mathsf{N},c} = !\,\mathsf{new}\ c'.\mathsf{out}(c,c').\mathsf{out}(c',\mathsf{senc}(n,k)) \ \mid\ \mathsf{new}\, n^\star.\mathsf{out}(c^\star,\mathsf{senc}(n^\star,k)).$$

3.2.2 Main result

We are now able to state our main result. We consider a signature $\Sigma = \Sigma_c \uplus \Sigma_d$ together with a convergent rewriting system $\mathcal{R}$, and a notion of messages $\mathcal{M}_\Sigma$ such that the theory $(\Sigma, \mathcal{R})$ is adequate w.r.t. $\mathcal{M}_\Sigma$. Given a simple process P, we note $\mathsf{Ch}(P)$ the set of public channel names occurring under a replication in P.

Theorem 3.2.1. Let P and Q be two simple protocols such that $\mathsf{Ch}(P) = \mathsf{Ch}(Q)$, and N be a set of names (intuitively those that we want to abstract away). We have that:

$$[\forall c \in \mathsf{Ch}(P).\ \overline{P}^{\mathsf{N},c} \approx \overline{Q}^{\mathsf{N},c}] \ \Rightarrow\ P \approx Q$$

Note that, in case $\mathsf{Ch}(P) \neq \mathsf{Ch}(Q)$, we trivially have that $P \not\approx Q$ since one process is able to emit on a channel whereas the other is not.

This theorem shows that whenever two processes are not in trace equivalence, then it is possible to find a witness of non-equivalence when nonces are replaced by constants provided that one basic process under a replication has been duplicated.

Example 3.2.3. Continuing the example developed in introduction of this chapter and pursued in Section 3.2.1, we consider

1. $P = !\mathsf{new}\ c'.\mathsf{out}(c,c').\mathsf{new}\ n_P.\mathsf{out}(c',\mathsf{senc}(n_P,k))$, and
2. $Q = !\mathsf{new}\ c'.\mathsf{out}(c,c').\mathsf{out}(c',\mathsf{senc}(n_Q,k))$.

Let $\mathsf{N} = \{n_P\}$. We have that:

1. $\overline{P}^c = !\mathsf{new}\ c'.\mathsf{out}(c,c').\mathsf{out}(c',\mathsf{senc}(n_P,k)) \ \mid\ \mathsf{new}\, n_P^\star.\mathsf{out}(c^\star,\mathsf{senc}(n_P^\star,k))$, and
2. $\overline{Q}^c = !\mathsf{new}\ c'.\mathsf{out}(c,c').\mathsf{out}(c',\mathsf{senc}(n_Q,k)) \ \mid\ \mathsf{out}(c^\star,\mathsf{senc}(n_Q,k))$.

Clearly $\overline{P}^c \not\approx \overline{Q}^c$ since an attacker can observe that $\overline{P}^c$ may send two distinct messages while $\overline{Q}^c$ cannot. Intuitively, the attack reflecting that $P \not\approx Q$ can be reflected in $\overline{P}^c \not\approx \overline{Q}^c$. Another choice for N is to consider the set $\{n_P, n_Q\}$ but this would lead exactly to the same result.

3.2.3 Proof

To establish our result, we first establish how to map traces from P to $\overline{P}^{\mathsf{N}}$. Given a simple process P, and a trace $(\mathsf{tr},\phi) \in \mathsf{trace}(P)$, we denote by $\rho^{P,\mathsf{N}}_{(\mathsf{tr},\phi)}$ the replacement that associates to each name $r \in \mathcal{N}$ generated during the execution under study and occurring in the frame ϕ, the name $n \in \mathsf{N}$ that occurs in the instruction $\mathsf{new}\ n$ of P and that is responsible of the generation of this fresh name. This amounts in losing freshness of all the $\mathsf{new}\ n$ instructions with $n \in \mathsf{N}$. Indeed all nonces induced by such an instruction are collapsed into a single nonce n. Our transformation is parametric in N: we may replace all new instructions or simply part of them. Note that, for simple processes, once (tr,ϕ) is fixed, this replacement is uniquely defined.

Lemma 3.2.1. Let P be a simple protocol, N be a set of names, and $(\mathsf{tr},\phi) \in \mathsf{trace}(P)$. We have that $(\mathsf{tr}, \phi\rho^{P,\mathsf{N}}_{(\mathsf{tr},\phi)}) \in \mathsf{trace}(\overline{P}^{\mathsf{N}})$.

Lemma 3.2.1 is a direct corollary of Lemma 3.2.2 which we state below. In the following, we will only consider theories adequate w.r.t. $\mathcal{M}_\Sigma$. Given a frame ϕ (resp. ψ) and a name r in ϕ (resp. ψ), let $n(r)$ be the nonce in P (resp. Q) such that r is an instance of $n(r)$ and let $c(r)$ be the channel of the protocol's branch which generated it. Actually, it can be computed as the channel on which r appeared first in $\mathsf{tr}\phi\downarrow$ (resp. $\mathsf{tr}\psi\downarrow$). We note $\mathcal{D}_\phi = \{r \in \phi \mid n(r) \in \mathsf{N}\}$ and, by extension, $n(A) = \{r \mapsto n(r) \mid r \in A\}$ for any $A \subseteq \mathcal{D}_\phi$. To each nonce $n \in \mathsf{N}$, we can associate a new name $n^\star$: we can then define the function $n^\star(\cdot)$ to be the function mapping any $r \in \mathcal{D}_\phi$ to $(n(r))^\star$. Similarly, $n^\star(A) = \{r \mapsto (n(r))^\star \mid r \in A\}$ for $A \subseteq \mathcal{D}_\phi$.

Lemma 3.2.2. We have the two following properties.

1. Let $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, $\mathcal{D}_\phi = \{r \in \phi \mid n(r) \in \mathsf{N}\}$ and $\rho_0 = n(D_\phi)$. Then $(\mathsf{tr}, \phi\rho_0) \in \mathsf{trace}(\overline{P}^{\mathsf{N}})$.
2. Moreover, let ch be a channel such that $\mathsf{tr} = \mathsf{tr}_1.\mathsf{out}(c, ch).\mathsf{tr}_2$, $\tilde{\mathcal{D}}_\phi = \{r \in \phi \mid n(r) \in \mathsf{N} \wedge c(r) = ch\}$ and $\rho = n(\mathcal{D}_\phi \setminus \tilde{\mathcal{D}}_\phi) \cup n^\star(\tilde{\mathcal{D}}_\phi)$. Then $(\mathsf{tr}^\star, \phi\rho) \in \mathsf{trace}(\overline{P}^{\mathsf{N},c})$, where $\mathsf{tr}^\star = \mathsf{tr}_1.\mathsf{tr}_2\{^{c^\star}/_{ch}\}$.

Proof. We first prove the second item. For any trace $\mathsf{tr} = \mathsf{tr}_1.\mathsf{out}(c, ch).\mathsf{tr}_2$, we note $\mathsf{tr}^\star = \mathsf{tr}_1.\mathsf{tr}_2\{^{c^\star}/_{ch}\}$. We define ρ' as:

$$\rho' = \{n \mapsto n^\star \mid n \in \mathsf{N} \wedge \text{"}\mathsf{new}\, n\text{" occurs in } B\}$$

where B is described in the definition of the transformation. We want to prove that if $P \xrightarrow{\tau.s} (\mathcal{P} \cup M, \phi)$ then $\overline{P}^{\mathsf{N},c} \xrightarrow{\bar{s}} (\bar{\mathcal{P}}\rho \cup \bar{M}^\star\rho\rho', \phi\rho)$ where s is a sequence of actions (visible *and* invisible), $\bar{s}$ is a subsequence of s with the same visible actions (*i.e.* only τ-actions can be deleted) except for *one* action $\mathsf{out}(c, ch)$ deleted and ch is substituted by $c^\star$, $\bar{\mathcal{P}}$ is $\mathcal{P}$ expurgated of the $\mathsf{new}\ n$ elements for $n \in \mathsf{N}$, M is either $\mathsf{new}\ c'.\mathsf{out}(c, c').B$, a process on channel ch (which results from the execution of $\mathsf{new}\ c'.\mathsf{out}(c, c').B$) or the 0 process; and $\bar{M}^\star$ is M expurgated of the $\mathsf{new}\ n$ elements for $n \in \mathsf{N}$; and ch replaced by $c^\star$. We proceed by induction on s.

Base case: if $s = \epsilon$: it holds by definition of $\overline{P}^{\mathsf{N},c}$ and ρ' ($\phi = \emptyset$ and nonces are not instantiated yet). We require one τ-action to unfold the branch $!\mathsf{new}\ c'.\mathsf{out}(c, c').B$ once in P.

Induction cases: now we consider the cases where $s = s'.a$ for some action a. We have that

$$P \xrightarrow{\tau.s'} (\mathcal{P}' \cup M', \phi') \xrightarrow{a} (\mathcal{P} \cup M, \phi).$$

By induction hypothesis, $\overline{P}^{\mathsf{N},c} \xrightarrow{\bar{s}'} (\bar{\mathcal{P}}'\rho \cup \bar{M}'^\star\rho\rho', \phi'\rho)$. We want to prove that in $\overline{P}^{\mathsf{N},c}$:

$$(\bar{\mathcal{P}}'\rho \cup \bar{M}'^\star\rho\rho', \phi'\rho) \xrightarrow{\bar{a}} (\bar{\mathcal{P}}\rho \cup \bar{M}^\star\rho\rho', \phi\rho).$$

- If $a = \tau$ and a corresponds to a replication: if there exists a process $!R$ in $\mathcal{P}'$ on which to apply the replication unfolding, then $!\bar{R}\rho \in \bar{\mathcal{P}}'\rho$, which directly leads to $(\bar{\mathcal{P}}'\rho \cup \bar{M}'^\star\rho\rho', \phi'\rho) \xrightarrow{\tau} (\bar{\mathcal{P}}\rho \cup \bar{M}^\star\rho\rho', \phi\rho)$.
- If $a = \tau$ and a corresponds to the restriction of a name $n \in \mathsf{N}$ which does not occur in M': then there exists Q such that $\mathsf{new}\ n.Q \in \mathcal{P}'$ and thus $\bar{Q}\rho \in \bar{\mathcal{P}}'\rho$ as $n \in \mathsf{N}$ and there exists r such that n is replaced by r in Q. As $r \in \mathcal{D}_\phi \setminus \tilde{\mathcal{D}}_\phi$ ($\mathsf{new}\ n$ does not occur in M' and thus $c(r) \neq ch$), $r\rho = n$ and thus $\bar{\mathcal{P}}'\rho = \bar{\mathcal{P}}\rho$. Hence we get that $(\bar{\mathcal{P}}'\rho \cup \bar{M}'^\star\rho\rho', \phi'\rho) \xrightarrow{\epsilon} (\bar{\mathcal{P}}\rho \cup \bar{M}^\star\rho\rho', \phi\rho)$.
- If $a = \tau$ and a corresponds to the restriction of a name $n \in \mathsf{N}$ which occurs in M': then there exists Q such that $\mathsf{new}\ n.Q = M'$ and thus $\bar{Q}^\star\rho\rho' = \bar{M}'^\star\rho\rho'$ as $n \in \mathsf{N}$ and there exists r such that n is replaced by r in Q. As $r \in \tilde{\mathcal{D}}_\phi$ ($\mathsf{new}\ n$ occurs in M' and thus $c(r) = ch$), $r\rho = n^\star$. Then $\mathcal{P}' = \mathcal{P}$ and $\bar{M}'^\star\rho\rho' = \bar{M}^\star\rho\rho'$ (as $\bar{M}'^\star\rho = \bar{M}^\star\rho$). Hence we get that $(\bar{\mathcal{P}}'\rho \cup \bar{M}'^\star\rho\rho', \phi'\rho) \xrightarrow{\epsilon} (\bar{\mathcal{P}}\rho \cup \bar{M}^\star\rho\rho', \phi\rho)$.
- If $a = \tau$ and a corresponds to the restriction of a name $n \notin \mathsf{N}$: in this case, $\mathsf{new}\ n$ appears in both $\mathcal{P}'$ and $\bar{\mathcal{P}}'\rho$ (or both M' and $\bar{M}'^\star\rho$) and is replaced by a nonce $r \notin \mathcal{D}_\phi$, leading to $(\bar{\mathcal{P}}'\rho \cup \bar{M}'^\star\rho\rho', \phi'\rho) \xrightarrow{\tau} (\bar{\mathcal{P}}\rho \cup \bar{M}^\star\rho\rho', \phi\rho)$.

- If $a = \mathsf{out}(ch', \mathsf{w})$ for some $ch' \neq ch$: $\mathsf{out}(ch', u).Q \in \mathcal{P}'$ for some term u and process Q implies $\mathsf{out}(ch', u\rho).\bar{Q}\rho \in \bar{\mathcal{P}}'\rho$ which ensures that $(\bar{\mathcal{P}}'\rho \cup \bar{M}'^{\star}\rho\rho', \phi'\rho) \xrightarrow{\mathsf{out}(ch', \mathsf{w})} (\bar{\mathcal{P}}\rho \cup \bar{M}^{\star}\rho\rho', \phi\rho)$.
- If $a = \mathsf{out}(ch, \mathsf{w})$: $M' = \mathsf{out}(ch, u).Q$ for some term u and process Q implies $\bar{M}'^{\star}\rho\rho' = \mathsf{out}(c^{\star}, u\rho\rho').\bar{Q}^{\star}\rho\rho'$ (as communications occur on $c^{\star}$ instead of ch in $\bar{M}'^{\star}$). As u cannot contain any non-instantiated nonce, $u\rho' = u$ and $u\rho\rho' = u\rho$ (as ρ and ρ' commute). Which leads us to $(\bar{\mathcal{P}}'\rho \cup \bar{M}'^{\star}\rho\rho', \phi'\rho) \xrightarrow{\mathsf{out}(c^{\star}, \mathsf{w})} (\bar{\mathcal{P}}\rho \cup \bar{M}^{\star}\rho\rho', \phi\rho)$.
- If $a = \mathsf{in}(ch', R)$ for some $ch' \neq ch$ and recipe R. Note that $(R\phi'\downarrow)\rho = R(\phi'\rho)\downarrow$ as $R\phi'\downarrow$ reduces to a message and thus contains no destructors. Then $\mathsf{in}(ch', u).Q \in \mathcal{P}'$, for some term u and process Q, implies $\mathsf{in}(ch', u\rho).\bar{Q}\rho \in \bar{\mathcal{P}}'\rho$ which ensures that, because there exists σ such that $R\phi'\downarrow = u\sigma$, $R\psi'\downarrow\rho = R(\psi\rho)\downarrow = (u\sigma)(\sigma\rho) = (u\sigma)\rho$ and thus the pattern matching is successful in $\mathcal{P}'$. Then, as in the case where $a = \mathsf{out}(ch', \mathsf{w})$ for some $ch' \neq ch$, we get that $(\bar{\mathcal{P}}'\rho \cup \bar{M}'^{\star}\rho\rho', \phi'\rho) \xrightarrow{\mathsf{in}(ch', R)} (\bar{\mathcal{P}}\rho \cup \bar{M}^{\star}\rho\rho', \phi\rho)$.
- If $a = \mathsf{in}(ch, R)$ for some recipe R. Here note that $(R\phi'\downarrow)\rho\rho' = R(\phi'\rho\rho')\downarrow$ as $R\phi'\downarrow$ reduces to a message and thus contains no destructors. Then, as in the case where $a = \mathsf{out}(ch, \mathsf{w})$, we get that $(\bar{\mathcal{P}}'\rho \cup \bar{M}'^{\star}\rho\rho', \phi'\rho) \xrightarrow{\mathsf{in}(c^{\star}, R)} (\bar{\mathcal{P}}\rho \cup \bar{M}^{\star}\rho\rho', \phi\rho)$.
- If $a = \mathsf{out}(c, ch)$: $M' = \mathsf{new}\ c'.\mathsf{out}(c, c').Q$ for some process Q implies $\bar{M}'^{\star}\rho\rho' = \bar{Q}^{\star}\rho\rho'$ and then $\bar{M}^{\star}\rho\rho' = \bar{Q}^{\star}\rho\rho'$. Hence $(\bar{\mathcal{P}}'\rho \cup \bar{M}'^{\star}\rho\rho', \phi'\rho) \xrightarrow{\epsilon} (\bar{\mathcal{P}}\rho \cup \bar{M}^{\star}\rho\rho', \phi\rho)$. Note that this action can only occur once in s, as channels are freshly generated and ch is thus unique.
- If $a = \mathsf{out}(c, ch')$ for some $ch' \neq ch$: $\mathsf{new}\ c'.\mathsf{out}(c, c').Q \in \mathcal{P}'$ for some process Q implies $\mathsf{new}\ c'.\mathsf{out}(c, c').\bar{Q}\rho \in \bar{\mathcal{P}}'\rho$, which leads us to
  $(\bar{\mathcal{P}}'\rho \cup \bar{M}'^{\star}\rho\rho', \phi'\rho) \xrightarrow{\mathsf{out}(c, ch)} (\bar{\mathcal{P}}\rho \cup \bar{M}^{\star}\rho\rho', \phi\rho)$.
- If $a = \mathsf{out}(c'', ch')$ for some $c'' \neq c$ and some ch' (necessarily $ch' \neq ch$ as generated channels are fresh): this case is treated as the previous one.
- If $a = \tau$ and a corresponds to a term evaluation $\mathsf{let}\ x = v\ \mathsf{in}\ P$, whether in a process in $\mathcal{P}'$ or in M', as messages are stable by renaming, we get the result, in the same way as for the inputs.

Thus we managed to prove that $\overline{P}^{\mathsf{N},c} \xrightarrow{\bar{s}} (\bar{\mathcal{P}}\rho \cup \bar{M}^{\star}\rho\rho', \phi\rho)$, which, by definition of $\bar{s}$ translate to $\overline{P}^{\mathsf{N},c} \xRightarrow{\mathsf{tr}^{\star}} (\bar{\mathcal{P}}\rho \cup \bar{M}^{\star}\rho\rho', \phi\rho)$ and proves that $(\mathsf{tr}^{\star}, \phi\rho) \in \mathsf{trace}(\overline{P}^{\mathsf{N},c})$.

The first item of the lemma can be seen as a special case of the previous proof where no action $\mathsf{out}(c, ch)$ ever occurs and we do not need to split the multiset of processes between $\mathcal{P}$ and M. Hence the lemma. □

Now, it remains to ensure that the disequality that is needed to witness the non-equivalence still remains, and this is the purpose of considering a fresh copy, namely $B^{\star}(c^{\star})$.

The idea is to show that a witness of non-equivalence for $P \not\approx Q$ can be converted into a witness of non-equivalence for $\overline{P}^{c} \not\approx \overline{Q}^{c}$ for at least one $c \in \mathsf{Ch}(P) = \mathsf{Ch}(Q)$. Due to the fact that we consider simple processes, three main cases may occur (the three other symmetric cases can be handled similarly). We have that $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, and

1. there exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ and two recipes R_1, R_2 such that $R_1\phi\downarrow$, $R_2\phi\downarrow$, $R_1\psi\downarrow$ and $R_2\psi\downarrow$ are messages; $R_1\phi\downarrow = R_2\phi\downarrow$ and $R_1\psi\downarrow \neq R_2\psi\downarrow$; or
2. there exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ and a recipe R such that $R\phi\downarrow$ is a message but $R\psi\downarrow$ is not; or
3. there exists no frame ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$.

Each case is proved separately, following the same lines. First, we easily conclude thanks to Lemma 3.2.1, in case $(\mathsf{tr}, \phi\rho^{P,\mathsf{N}}_{(\mathsf{tr},\phi)})$ is still a witness of non-equivalence. This roughly means that we do not even need the fresh copy to exhibit the non-equivalence. Otherwise, we need to maintain a disequality to ensure that the distinguishing test will not hold on the Q side. Since we consider adequate theories, we know that this disequality can be maintained through the use of two distinct names. This is exactly why a fresh copy is needed. The other cases can be handled similarly.

Theorem 3.2.1. Let P and Q be two simple protocols such that $\mathsf{Ch}(P) = \mathsf{Ch}(Q)$, and N be a set of names (intuitively those that we want to abstract away). We have that:

$$[\forall c \in \mathsf{Ch}(P).\ \overline{P}^{\mathsf{N},c} \approx \overline{Q}^{\mathsf{N},c}] \ \Rightarrow\ P \approx Q$$

Proof. Let us assume there exists a witness of non-equivalence $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$. Three main cases can occur:

1. there exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ and two recipes R_1, R_2 such that $R_1\phi\downarrow$, $R_2\phi\downarrow$, $R_1\psi\downarrow$ and $R_2\psi\downarrow$ are messages; $R_1\phi\downarrow = R_2\phi\downarrow$ and $R_1\psi\downarrow \neq R_2\psi\downarrow$;
2. or there exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ and a recipe R such that $R\phi\downarrow$ is a message but $R\psi\downarrow$ is not;
3. or, finally, there exists no frame ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$.

Note that the remaining symmetric cases are handled by considering a witness $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ instead, as P and Q are both simple. We will deal with each case separately, with the same intermediate goal: define a renaming ρ on D_ψ such that any test failing in ψ still fails $\psi\rho$ while the successful tests in ϕ remain so; then translate it into a valid trace of $\overline{P}^{\mathsf{N},c}$ for some $c \in \mathsf{Ch}(P)$.

Case 1: Let us examine $R_1\psi\downarrow$ and $R_2\psi\downarrow$. If the two terms do not share the same constructors, then for any renaming ρ, $R_1(\psi\rho)\downarrow \neq R_2(\psi\rho)\downarrow$, while for any renaming ρ', $R_1(\phi\rho')\downarrow = R_2(\phi\rho')\downarrow$ (as the constructors are left unchanged, because every term is a message). Now, if the two terms share the same constructors, there must exist a leaf position p in them such that $R_1\psi\downarrow|_p \neq R_2\psi\downarrow|_p$. Let us call t and s these terms respectively. If s or t is *not* an element of $\mathcal{D}_\psi$, then $s\rho \neq t\rho$ for any ρ with $dom(\rho) = \mathcal{D}_\psi$. As in the previous case, we get that $R_1(\psi\rho)\downarrow \neq R_2(\psi\rho)\downarrow$, while $R_1(\phi\rho')\downarrow = R_2(\phi\rho')\downarrow$ for any renaming ρ'. Else, assume $s = r_1$ and $t = r_2$ are two nonces of $\mathcal{D}_\psi$ such that $n(r_1) = n_1 \in \mathsf{N}$ (resp. $n(r_2) = n_2 \in \mathsf{N}$). If $n_1 \neq n_2$, consider the renaming $\rho_0^Q = \{r \mapsto n(r) | r \in \mathcal{D}_\psi\}$. Then $s\rho_0^Q \neq t\rho_0^Q$ and we get that $R_1(\psi\rho_0^Q)\downarrow \neq R_2(\psi\rho_0^Q)\downarrow$. By Lemma 3.2.2, $(\mathsf{tr}, \psi\rho_0^Q) \in \mathsf{trace}(\overline{Q}^{\mathsf{N}})$. Similarly, by defining $\rho_0^P = \{r \mapsto n(r) \mid r \in \mathcal{D}_\phi\}$, $(\mathsf{tr}, \phi\rho_0^P) \in \mathsf{trace}(\overline{P}^{\mathsf{N}})$. and $R_1(\phi\rho_0^P)\downarrow = R_2(\phi\rho_0^P)\downarrow$. Hence we get a witness of non-equivalence between $\overline{P}^{\mathsf{N}}$ and $\overline{Q}^{\mathsf{N}}$, which can translate into a witness between $\overline{P}^{\mathsf{N},c}$ and $\overline{Q}^{\mathsf{N},c}$ for any $c \in \mathsf{Ch}(P)$.

Else, if $n(r_1) = n(r_2) = n$, we need to be more precise to define a proper ρ. Let $\mathsf{out}(c, ch)$ be the action of tr such that $\mathsf{tr} = \mathsf{tr}_1.\mathsf{out}(c, ch).\mathsf{tr}_2$ and $c(r_2) = ch$. Let $\tilde{\mathcal{D}}_\psi = \{r \in \psi \mid n(r) \in \mathsf{N} \wedge c(r) = ch\}$ and $\tilde{\mathcal{D}}_\phi = \{r \in \phi \mid n(r) \in \mathsf{N} \wedge c(r) = ch\}$. $r_1 \in \mathcal{D}_\psi \setminus \tilde{\mathcal{D}}_\psi$ but $r_2 \in \tilde{\mathcal{D}}_\psi$ Consider now $\rho_Q = n(\mathcal{D}_\psi \setminus \tilde{\mathcal{D}}_\psi) \cup n^\star(\tilde{\mathcal{D}}_\psi)$. In particular, $r_1\rho_Q = n$ while $r_2\rho_Q = n^\star$. Then $s\rho_Q \neq t\rho_Q$ and we get that $R_1(\psi\rho_Q)\downarrow \neq R_2(\psi\rho_Q)\downarrow$ and Lemma 3.2.2 ensures $(\mathsf{tr}^\star, \psi\rho_Q) \in \mathsf{trace}(\overline{Q}^{\mathsf{N},c})$. Similarly, by defining $\rho_P = n(\mathcal{D}_\phi \setminus \tilde{\mathcal{D}}_\phi) \cup n^\star(\tilde{\mathcal{D}}_\phi)$, Lemma 3.2.2 ensures $(\mathsf{tr}^\star, \phi\rho_P) \in \mathsf{trace}(\overline{P}^{\mathsf{N},c})$ and $R_1(\phi\rho_P)\downarrow = R_2(\phi\rho_P)\downarrow$ (only equalities have been introduced by removing the name restriction in P). Hence we get a witness of non-equivalence between $\overline{P}^{\mathsf{N},c}$ and $\overline{Q}^{\mathsf{N},c}$

Case 2: Because $R\psi\downarrow$ is not a message and our signature is adequate (see Definition 3.1.1), there must exist $a, b \in \mathcal{N}$ such that $a \neq b$ and for any renaming $\sigma : \mathcal{N} \to \mathcal{N}$, $a\sigma \neq b\sigma \Rightarrow t\sigma\downarrow \notin \mathcal{M}_\Sigma$. If $a \notin \mathcal{D}_\psi$ or $b \notin \mathcal{D}_\psi$, consider the renaming $\rho_0^Q = \{r \mapsto n(r) \mid r \in \mathcal{D}_\psi\}$: as $a\rho_0^Q = a$ and $n(r) \neq a$ for any $r \in \mathcal{D}_\psi$, $R(\psi\rho_0^Q)\downarrow$ is still not a message. On the other hand, if $\rho_0^P = \{r \mapsto n(r) \mid r \in \mathcal{D}_\phi\}$, as $R\phi\downarrow$ is a message, $R\phi\downarrow\rho_0^P = R(\phi\rho_0^P)\downarrow$ is a message. Hence, Lemma 3.2.2 ensures $(\mathsf{tr}, \phi\rho_0^P) \in \mathsf{trace}(\overline{P}^{\mathsf{N}})$ while $(\mathsf{tr}, \psi\rho_0^Q) \notin \mathsf{trace}(\overline{Q}^{\mathsf{N}})$, leading to a witness of non-equivalence between $\overline{P}^{\mathsf{N}}$ and $\overline{Q}^{\mathsf{N}}$.

Else, assume $a = r_1$ and $b = r_2$ are two nonces in $\mathcal{D}_\psi$. If $n(r_1) \neq n(r_2)$, $r_1\rho_0^Q \neq r_2\rho_0^Q$ and we can apply the same exact reasoning as before. So let us consider the case where $n(r_1) = n(r_2) = n$. Let $\mathsf{out}(c, ch)$ be the action of tr such that $\mathsf{tr} = \mathsf{tr}_1.\mathsf{out}(c, ch).\mathsf{tr}_2$ and $c(r_2) = ch$. Let $\tilde{\mathcal{D}}_\psi = \{r \in \psi \mid n(r) \in \mathsf{N} \wedge c(r) = ch\}$ and $\tilde{\mathcal{D}}_\phi = \{r \in \phi \mid n(r) \in \mathsf{N} \wedge c(r) = ch\}$. $r_1 \in \mathcal{D}_\psi \setminus \tilde{\mathcal{D}}_\psi$ but $r_2 \in \tilde{\mathcal{D}}_\psi$ Consider now $\rho_Q = n(\mathcal{D}_\psi \setminus \tilde{\mathcal{D}}_\psi) \cup n^\star(\tilde{\mathcal{D}}_\psi)$. In particular, $r_1\rho_Q = n$ while $r_2\rho_Q = n^\star$. Definition 3.1.1 ensures $R(\psi\rho_0^Q)\!\downarrow$ is still not a message. On the other hand, if $\rho_P = n(\mathcal{D}_\phi \setminus \tilde{\mathcal{D}}_\phi) \cup n^\star(\tilde{\mathcal{D}}_\phi)$, as $R\phi\!\downarrow$ is a message, $R\phi\!\downarrow\rho_P = R(\phi\rho_P)\!\downarrow$ is a message. Hence, Lemma 3.2.2 ensures $(\mathsf{tr}^\star, \phi\rho_P) \in \mathsf{trace}(\overline{P}^{\mathsf{N},c})$ while $(\mathsf{tr}^\star, \psi\rho_Q) \in \mathsf{trace}(\overline{Q}^{\mathsf{N},c})$, leading to a witness of non-equivalence between $\overline{P}^{\mathsf{N},c}$ and $\overline{Q}^{\mathsf{N},c}$.

Case 3: if tr ends with an output $\mathsf{out}(c, \mathsf{w})$ such that $\mathsf{w}\psi$ is not a message, we can define ρ_Q and ρ_P as in case 2 and obtain a witness of non-equivalence. Similarly, if tr ends with an input or output $\mathsf{out}(c, \mathsf{w})$ which cannot be executed in Q because a let action did not reduce to a message, we can define ρ_Q and ρ_P as in case 2 and obtain a witness of non-equivalence. Consider now the subcase where $\mathsf{tr} = \mathsf{tr}'.\mathsf{in}(c, R)$ for some tr' such that $(\mathsf{tr}', \phi) \in \mathsf{trace}(P)$ and $(\mathsf{tr}', \psi) \in \mathsf{trace}(Q)$ for some frame ψ. Because P and Q are both simple protocols, there exists a unique term u_P (resp. at most one term u_Q) in the multiset $\mathcal{P}$ (resp. $\mathcal{Q}$) of processes from the execution of tr' in P (resp. in Q) such that $\mathsf{in}(c, u_P).M \in \mathcal{P}$ for some M (resp. $\mathsf{in}(c, u_Q).N \in \mathcal{Q}$ for some N). Moreover, there exists σ_P such that $R\phi\!\downarrow = u_P\sigma_P$ while there is no σ such that $R\psi\!\downarrow = u_Q\sigma$. As before, we consider the renamings $\rho_0^Q = n(\mathcal{D}_\psi)$ and $\rho_0^P = n(\mathcal{D}_\phi)$. As $(\mathsf{tr}, \phi\rho_0^P) \in \mathsf{trace}(\overline{P}^{\mathsf{N}})$ and $(\mathsf{tr}, \psi\rho_0^Q) \in \mathsf{trace}(\overline{Q}^{\mathsf{N}})$ by Lemma 3.2.2, if there exists no σ such that $u_Q\rho_0^Q\sigma = R\psi\!\downarrow\rho_0^Q$, tr is a witness of non-equivalence between $\overline{P}^{\mathsf{N}}$ and $\overline{Q}^{\mathsf{N}}$ and we are done. So let us then assume there exists σ_0 such that $u_Q\rho_0^Q\sigma_0 = R\psi\!\downarrow\rho_0^Q$ while $u_Q\sigma \neq R\psi\!\downarrow$ for every σ. There exist two leaves with positions p_1 and p_2 in $R\psi\!\downarrow$ which correspond to positions below variables in u_Q such that $R\psi\!\downarrow|_{p_1} \neq R\psi\!\downarrow|_{p_2}$ but $R(\psi\rho_0^Q)\!\downarrow|_{p_1} = R(\psi\rho_0^Q)\!\downarrow|_{p_2}$ and $R\psi\!\downarrow|_{p_1} = r_1$ and $R\psi\!\downarrow|_{p_2} = r_2$ such that $n(r_1) = n(r_2) = n \in \mathsf{N}$. As repeatedly before, let $\mathsf{out}(c, ch)$ be the action of tr such that $\mathsf{tr} = \mathsf{tr}_1.\mathsf{out}(c, ch).\mathsf{tr}_2$ and $c(r_2) = ch$. Let $\tilde{\mathcal{D}}_\psi = \{r \in \psi \mid n(r) \in \mathsf{N} \wedge c(r) = ch\}$ and $\tilde{\mathcal{D}}_\phi = \{r \in \phi \mid n(r) \in \mathsf{N} \wedge c(r) = ch\}$. $r_1 \in \mathcal{D}_\psi \setminus \tilde{\mathcal{D}}_\psi$ but $r_2 \in \tilde{\mathcal{D}}_\psi$ Consider now $\rho_Q = n(\mathcal{D}_\psi \setminus \tilde{\mathcal{D}}_\psi) \cup n^\star(\tilde{\mathcal{D}}_\psi)$. In particular, $r_1\rho_Q = n$ while $r_2\rho_Q = n^\star$. As $R\psi\!\downarrow$ is a message (by virtue of our semantics), $R\psi\!\downarrow\rho_Q = R(\psi\rho_Q)\!\downarrow$ and now $R(\psi\rho_Q)\!\downarrow|_{p_1} \neq R(\psi\rho_Q)\!\downarrow|_{p_2}$. As such, $u_Q\rho_Q\sigma \neq R\psi\rho_Q\!\downarrow$ for any σ. By defining $\rho_P = n(\mathcal{D}_\phi \setminus \tilde{\mathcal{D}}_\phi) \cup n^\star(\tilde{\mathcal{D}}_\phi)$, as $R\phi\!\downarrow$ is a message, $R\phi\!\downarrow\rho_P = R(\phi\rho_P)\!\downarrow$ is a message. Hence, Lemma 3.2.2 ensures $(\mathsf{tr}^\star, \phi\rho_P) \in \mathsf{trace}(\overline{P}^{\mathsf{N},c})$ while $(\mathsf{tr}^\star, \psi) \notin \mathsf{trace}(\overline{Q}^{\mathsf{N},c})$ for any ψ, leading to a witness of non-equivalence between $\overline{P}^{\mathsf{N},c}$ and $\overline{Q}^{\mathsf{N},c}$. □

3.3 Scope of our result

In this section, we explain why we need to assume simple processes and adequate theories and we discuss which class of protocols and primitives can be covered.

3.3.1 Simple processes

Simple processes are really necessary for our simplification result to hold. We provide below a small counter example to our result for non simple processes.

Example 3.3.1. We consider symmetric encryption and pairs as in Example 2.1.1 with $\mathsf{ok} \in \Sigma_0$. We define the two following processes.

$$
\begin{array}{rcll}
P & = & !\,\mathsf{new}\,c.\mathsf{out}(c_1, c).\mathsf{new}\,n.\mathsf{out}(c, \mathsf{senc}(n, k)) & (1)\\
 & | & !\,\mathsf{new}\,c.\mathsf{out}(c_2, c).\mathsf{in}(c, \langle \mathsf{senc}(x, k), \mathsf{senc}(x, k), \mathsf{senc}(y, k)\rangle).\mathsf{out}(c, \mathsf{ok}) & (2)\\
 & | & !\,\mathsf{new}\,c.\mathsf{out}(c_2, c).\mathsf{in}(c, \langle \mathsf{senc}(x, k), \mathsf{senc}(y, k), \mathsf{senc}(x, k)\rangle).\mathsf{out}(c, \mathsf{ok}) & (3)\\
 & | & !\,\mathsf{new}\,c.\mathsf{out}(c_2, c).\mathsf{in}(c, \langle \mathsf{senc}(y, k), \mathsf{senc}(x, k), \mathsf{senc}(x, k)\rangle).\mathsf{out}(c, \mathsf{ok}) & (4)\\
Q & = & !\,\mathsf{new}\,c.\mathsf{out}(c_1, c).\mathsf{new}\,n.\mathsf{out}(c, \mathsf{senc}(n, k)) & \\
 & | & !\,\mathsf{new}\,c.\mathsf{out}(c_2, c).\mathsf{in}(c, \langle \mathsf{senc}(x, k), \mathsf{senc}(y, k), \mathsf{senc}(z, k)\rangle).\mathsf{out}(c, \mathsf{ok}). &
\end{array}
$$

Intuitively P expects a list of three ciphertexts among which two must be identical, while Q expects any three ciphertexts. The process Q is simple but P is *not* since several processes in parallel proceed on channel c_2. We have that $P \not\approx Q$: it is possible using (1) to generate distinct ciphertexts, concatenate them, and send the resulting message on c_2. This message will not be accepted in P, but it will be accepted in Q.

Now, consider the process $\overline{P}^{c_1}$ and $\overline{Q}^{c_1}$ with $\mathsf{N} = \{n\}$, that is the processes obtained by applying our transformation on channel c_1 (the only branch that contains nonce generation) with the goal of getting rid of the instruction $\mathsf{new}\, n$ on both sides. We obtain:

$$\begin{array}{rcl}
\overline{P}^{c_1} & = & !\,\mathsf{new}\, c.\mathsf{out}(c_1, c).\mathsf{out}(c, \mathsf{senc}(n, k)) \\
& | & \mathsf{new}\, n^\star.\, \mathsf{out}(c^\star, \mathsf{senc}(n^\star, k)) \\
& | & !\,\mathsf{new}\, c.\mathsf{out}(c_2, c).\mathsf{in}(c, \langle \mathsf{senc}(x, k), \mathsf{senc}(x, k), \mathsf{senc}(y, k)\rangle).\mathsf{out}(c, \mathsf{ok}) \\
& | & !\,\mathsf{new}\, c.\mathsf{out}(c_2, c).\mathsf{in}(c, \langle \mathsf{senc}(x, k), \mathsf{senc}(y, k), \mathsf{senc}(x, k)\rangle).\mathsf{out}(c, \mathsf{ok}) \\
& | & !\,\mathsf{new}\, c.\mathsf{out}(c_2, c).\mathsf{in}(c, \langle \mathsf{senc}(y, k), \mathsf{senc}(x, k), \mathsf{senc}(x, k)\rangle).\mathsf{out}(c, \mathsf{ok}) \\
\overline{Q}^{c_1} & = & !\,\mathsf{new}\, c.\mathsf{out}(c_1, c).\mathsf{out}(c, \mathsf{senc}(n, k)) \\
& | & \mathsf{new}\, n^\star.\, \mathsf{out}(c^\star, \mathsf{senc}(n^\star, k)) \\
& | & !\,\mathsf{new}\, c.\mathsf{out}(c_2, c).\mathsf{in}(c, \langle \mathsf{senc}(x, k), \mathsf{senc}(y, k), \mathsf{senc}(z, k)\rangle).\mathsf{out}(c, \mathsf{ok}).
\end{array}$$

It is quite easy to see that the witness of non-equivalence given above is not a valid one anymore. Actually, we have that $\overline{P}^{c_1}$ and $\overline{Q}^{c_1}$ are in trace equivalence since only two distinct ciphertexts may be produced.

Note that it is easy to express standard protocols as simple processes. As explained previously, encoding security protocols as simple processes is a good practice, and gives power to the attacker. However, it prevents the modelling of unlinkability properties.

3.3.2 Adequate theories

The fact that we consider adequate theories may seem to be a proof artefact. We could probably go beyond adequate theories, but this would be at the price of considering a more complex transformation, and in particular additional constants. We provide below an example of a theory that reflects the same kind of issues as the ones illustrated by the processes presented in Example 3.3.1.

Example 3.3.2. In addition to the signature introduced in Example 2.1.1, we consider an additional destructor symbol g together with the following rewriting rules:

$$\begin{array}{rcl}
\mathsf{g}(\langle \mathsf{senc}(x, z), \mathsf{senc}(x, z), \mathsf{senc}(y, z)\rangle) & \to & \mathsf{ok} \\
\mathsf{g}(\langle \mathsf{senc}(x, z), \mathsf{senc}(y, z), \mathsf{senc}(x, z)\rangle) & \to & \mathsf{ok} \\
\mathsf{g}(\langle \mathsf{senc}(y, z), \mathsf{senc}(x, z), \mathsf{senc}(x, z)\rangle) & \to & \mathsf{ok}
\end{array}$$

Assume for instance that $\mathcal{M}_\Sigma$ is $\mathcal{M}_c = \mathcal{T}(\Sigma_c, \mathcal{N})$ the set of all ground constructor terms. The resulting theory is not adequate. For instance, we have that the term $t = \mathsf{g}(\langle \mathsf{senc}(n_1, k), \mathsf{senc}(n_2, k), \mathsf{senc}(n_3, k)\rangle)$ is in normal form and not a message. However, any renaming ρ that preserves distinctness between only two names among n_1, n_2, n_3, will be such that $t\rho{\downarrow} \in \mathcal{M}_\Sigma$. This yields a counter-example to our result, illustrated by the two following processes.

$$\begin{array}{rcl}
P' & = & !\,\mathsf{new}\, c.\mathsf{out}(c_1, c).\mathsf{new}\, n.\mathsf{out}(c, \mathsf{senc}(n, k)) \\
& | & \mathsf{in}(c_2, \langle \mathsf{senc}(x_1, k), \mathsf{senc}(x_2, k), \mathsf{senc}(x_3, k)\rangle). \\
& & \qquad \mathsf{let}\ y = \mathsf{g}(\langle \mathsf{senc}(x_1, k), \mathsf{senc}(x_2, k), \mathsf{senc}(x_3, k)\rangle)\ \mathsf{in}\ \mathsf{out}(c_2, y). \\
Q' & = & !\,\mathsf{new}\, c.\mathsf{out}(c_1, c).\mathsf{new}\, n.\mathsf{out}(c, \mathsf{senc}(n, k)) \\
& | & \mathsf{in}(c_2, \langle \mathsf{senc}(x_1, k), \mathsf{senc}(x_2, k), \mathsf{senc}(x_3, k)\rangle).\mathsf{out}(c_2, \mathsf{ok}).
\end{array}$$

The process P' expects three ciphertexts and returns the result of applying g to them while Q' directly returns ok. For the same reasons as those explained in Example 3.3.1, we have that $P' \not\approx Q'$ whereas $\overline{P'}^{c_1} \approx \overline{Q'}^{c_1}$.

The equational theory above is contrived, and actually most of the equational theories useful to model cryptographic protocols can be shown to be adequate. An example of a non-adequate theory is tdcommit as described in [39] for modeling trapdoor commitment schemes, which does not fit the structure of our rules.

3.3.3 Is our abstraction precise enough?

Abstracting nonces with constants (as done in Theorem 3.2.1) may introduce false attacks. A typical case is when protocols make use of temporary secrets.

Example 3.3.3. Consider the signature described in Example 2.1.1. Let P and Q be:

$$\begin{aligned} P &= !\,\mathsf{new}\, c'.\mathsf{out}(c,c').\mathsf{in}(c',x).\mathsf{new}\, n.\mathsf{out}(c',\mathsf{senc}(\mathsf{ok},n)).\\ &\qquad\qquad \mathsf{let}\ y = \mathsf{sdec}(x,n)\ \mathsf{in}\ \mathsf{out}(c',y);\\ Q &= !\,\mathsf{new}\, c'.\mathsf{out}(c,c').\mathsf{in}(c',x).\mathsf{new}\, n.\mathsf{out}(c',n). \end{aligned}$$

The two processes are in equivalence: $P \approx Q$. Now, consider the processes $\overline{P}^c$ and $\overline{Q}^c$ with $\mathsf{N} = \{n\}$, that is, the processes obtained by applying our transformation on channel c to get rid of the fresh nonces.

$$\begin{aligned} \overline{P}^c &= !\,\mathsf{new}\, c'.\mathsf{out}(c,c').\mathsf{in}(c',x).\mathsf{out}(c',\mathsf{senc}(\mathsf{ok},n)).\\ &\qquad\qquad \mathsf{let}\ y = \mathsf{sdec}(x,n)\ \mathsf{in}\ \mathsf{out}(c',y)\\ &\quad \mid \mathsf{in}(c^\star,x).\mathsf{out}(c^\star,\mathsf{senc}(\mathsf{ok},n^\star)).\mathsf{let}\ y = \mathsf{sdec}(x,n^\star)\ \mathsf{in}\ \mathsf{out}(c^\star,y) \end{aligned}$$

$\overline{Q}^c$ is defined similarly. It is easy to notice that the output of the constant ok is now reachable, yielding $\overline{P}^c \not\approx \overline{Q}^c$.

Our transformation may in theory also introduce false attacks for protocols without temporary secrets. In this section, we review several (secure) protocols of the literature and study whether a false attack is introduced by our transformation. To perform this analysis we rely on the ProVerif tool. For each protocol, we first consider a scenario with honest agents only as for the Denning-Sacco protocol (Chapter 2). We then consider a richer scenario where honest agents are also willing to engage communications with a dishonest agent. In each case, we check whether ProVerif is able to establish:

1. the equivalence between the original processes (left column);
2. all the equivalences obtained after getting rid of all the nonces using our transformation (right column).

The results are reported on the table below: a ✓ means that ProVerif succeeded and a ✗ means that it failed. Actually, on most of the protocols/scenarios we have considered, our abstraction does not introduce any false attack. ProVerif models of our experiments are available online at `http://www.lsv.ens-cachan.fr/~chretien/prot.tar`.

Protocol name	original (with nonces)	our transformation (no nonce)
YAHALOM (corrected version)		
- simple scenario	✓	✓
- with a dishonest agent	✓	✓
OTWAY-REES		
- simple scenario	✓	✓
- with a dishonest agent	✓	✓
KAO-CHOW (tagged version)		
- simple scenario	✓	✓
- with a dishonest agent	✓	✓
NEEDHAM-SCHROEDER-LOWE		
- simple scenario (secrecy of N_a)	✓	✗
- simple scenario (secrecy of N_b)	✓	✓
- with a dishonest agent (secrecy of N_b)	✓	✓
DENNING-SACCO (asymmetric)		
- simple scenario	✓	✓
- with a dishonest agent	✓	✓

Needham Schroeder Lowe protocol. We briefly comment on the false attack introduced by our transformation on the Needham Schroeder Lowe protocol.

1. $A \to B$: $\{A, N_a\}_{\mathsf{pub}(B)}$	1. $I(A) \to B$: $\{A, N_i\}_{\mathsf{pub}(B)}$
2. $B \to A$: $\{N_a, N_b, B\}_{\mathsf{pub}(A)}$	2. $B \to I(A)$: $\{N_i, N_b, B\}_{\mathsf{pub}(A)}$
3. $A \to B$: $\{N_b\}_{\mathsf{pub}(B)}$	3. $I(A) \to B$: $\{N_b\}_{\mathsf{pub}(B)}$

The protocol is given on the left, and the (false) attack depicted on the right. This attack scenario (and more precisely step 3 of this scenario) is only possible when nonces are abstracted away with constants. Indeed, the attacker will not be able to decrypt the message $\{N_i, N_b, B\}_{\mathsf{pub}(A)}$ he has received to retrieve the nonce N_b. Instead he will simply replay an old message coming from a previous honest session between A and B. Since nonces have been replaced by constants, B will accept this old message, and will assume that N_i is a secret shared between A and B, while N_i is known by the attacker. Unfortunately, this abstraction does not seem to help ProVerif prove the security of new protocols. Nonetheless it can still be used as a proof technique to prove the security of protocols in classes defined in [23] and [24].

3.4 Conclusion

Our simplification result allows to soundly reduce the equivalence of processes with nonces to the equivalence of processes without nonces. This can be seen as a proof technique which removes a source of unboundedness problematic when trying to automatically check for trace equivalence. For example for tagged simple protocols with symmetric encryption, the resulting protocols fall in the decidable class of Chapter 4. Similarly, we could use the decidability result of Chapter 7 for ping-pong protocols with one variable per transition.

Our result assumes protocols to be simple processes. Otherwise, to prevent some transition, it could be necessary to maintain several disequalities. We plan to go slightly beyond simple processes and simply require some form of determinacy. More generally, we plan to study whether such a reduction result can be obtained for more general processes which include else branches to output errors messages, that is, study whether it is possible to compute a bound on the number of fresh copies from the structure of the processes.

Regarding adequate theories, we believe that our criterion is general enough to capture even more theories like exclusive or, or other theories with an associative and commutative operator. This would however require to extend our formalism to arbitrary terms (not just destructor/constructor theories).

Chapter 4

Well-typed executions

The subject of this chapter is a simplification result, that reduces the search space for attacks: if there is an attack, then there exists a well-typed attack. More formally, we show that if there is a witness (*i.e.* a trace) that $P \not\approx Q$ then there exists a witness which is well-typed w.r.t. P or Q, provided that P and Q are determinate processes (see Definition 2.4.3). This typing result holds for an unbounded number of sessions and an unbounded number of nonces, that is, it holds even if P and Q contain arbitrary replications and NEW operations. It holds for any typing system provided that any two unifiable encrypted subterms of P (or Q) are of the same type. It is then up to the user to adjust the typing system such that this hypothesis holds for the protocols under consideration.

As an application, we consider the class of tagged protocols introduced by Blanchet and Podelski [15] and defined in Chapter 2. An easy way to achieve this in practice is by labelling encryption and is actually a good protocol design principle [3, 49].

Interestingly, the proof of our main typing result involves providing a new decision procedure for trace equivalence in the case of a bounded number of sessions. This is a key intermediate result of our proof. Trace equivalence was already shown to be decidable for a bounded number of sessions (*e.g.* [65, 22]) but we propose a novel decision procedure that further provides a *well-typed* witness whenever the two processes are not in trace equivalence.

Our proof technique is inspired from the approach developed by Arapinis *et al* [6] for bounding the size of messages of an attack for the reachability case. Specifically, they show for some class of tagged protocols, that whenever there is an attack, there is a well-typed attack (for a particular typing system). We somehow extend their approach to trace equivalence and more general typing systems for symmetric encryption only, but without asymmetric encryption, signatures or hash functions.

4.1 Existence of a well-typed witness of non-equivalence

We consider the formal model defined in Chapter 2 with a restricted signature. More precisely, from this chapter onward, we will consider the signature

$$\Sigma = \{\mathsf{senc},\ \mathsf{sdec},\ \langle\ \rangle,\ \mathsf{proj}_1, \mathsf{proj}_2\} \uplus \Sigma_0$$

where Σ_0 is a set of atomic data, representing pairing and symmetric encryption as defined in Example 2.1.1. We also do not use evaluation in our processes, *i.e.* no let, and instead rely only on input filtering to model decryption operations. Moreover, we will consider only encryption with atomic keys, meaning that the set $\mathcal{M}$ of messages will be defined by $\mathcal{M} = \mathcal{M}_{\mathsf{atomic}}$. Note that this decision increases the power of the attacker as it lets her distinguish between situations she could not before, as described in Example 2.4.2.

We can now present our first main contribution: a typing result that reduces the search space for attacks. We first explain these hypotheses and then we state our general simplification result (see Theorem 4.1.1). The proof of

$$\mathsf{in}(c,u).P \cup \mathcal{P} \xrightarrow{\mathsf{in}(c,u)}_s P \cup \mathcal{P} \qquad !P \cup \mathcal{P} \xrightarrow{\tau}_s P' \cup !P \cup \mathcal{P}$$
$$\mathsf{out}(c,u).P \cup \mathcal{P} \xrightarrow{\mathsf{out}(c,u)}_s P \cup \mathcal{P} \qquad \mathsf{new}\, n.P \cup \mathcal{P} \xrightarrow{\tau}_s P\{^{n'}/_{n}\} \cup \mathcal{P}$$
$$\mathsf{new}\, c'.\mathsf{out}(c,c').P \cup \mathcal{P} \xrightarrow{\mathsf{out}(c,ch_i)}_s P\{^{ch_i}/_{c'}\} \cup \mathcal{P}$$

Figure 4.1: Symbolic semantics of protocols

this simplification result involves providing a novel decision procedure for trace equivalence in the case of a bounded number of sessions. The novelty of this decision procedure, in comparison to the existing ones, is to provide a well-typed witness whenever the two processes are not in trace equivalence. This key intermediate result is stated in Proposition 4.1.1.

4.1.1 Well-typed trace

Whether or not a trace is well-typed is defined w.r.t. the set of *symbolic traces* of a protocol. Formally, we define $\xrightarrow{\mathsf{tr}_s}_s$ to be the transitive closure of the relation $\xrightarrow{\alpha_s}_s$ defined between processes in Figure 4.1.1: where P' is equal to P up to renaming of variables that do not occur yet in the trace with fresh ones (of the same type), n' is a fresh name (of the same type as n), and ch_i is the "next" fresh channel name available in $\mathcal{C}h^{\mathsf{fresh}}$.

Then, the set of *symbolic traces* $\mathsf{trace}_s(P)$ of a protocol P is defined as follows:

$$\mathsf{trace}_s(P) = \{\mathsf{tr}_s \mid P \xrightarrow{\mathsf{tr}_s}_s Q \text{ for some } Q\ \}.$$

Intuitively, the symbolic traces are simply all possible traces before instantiation of the variables, with some renaming to avoid unwanted captures.

Example 4.1.1. Let $P_1 = \mathsf{in}(c,x).!\mathsf{new}\, k.\, \mathsf{in}(c, \mathsf{enc}(\langle x, y\rangle, k))$. We have that:

$$\mathsf{tr}_s = \mathsf{in}(c,x).\mathsf{in}(c, \mathsf{enc}(\langle x, y_1\rangle, k_1)).\mathsf{in}(c, \mathsf{enc}(\langle x, y_2\rangle), k_2) \in \mathsf{trace}_s(P_1)$$

Indeed, the variable x is bound before replication.

As stated in the lemma below, any concrete trace is the instance of a symbolic trace.

Lemma 4.1.1. Let P be a protocol and $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$. We have that $\mathsf{tr}\phi{\downarrow} = \mathsf{tr}_s\sigma$ for some $\mathsf{tr}_s \in \mathsf{trace}_s(P)$ and some substitution σ.

A well-typed trace is simply a trace that is well-typed w.r.t. one of the symbolic traces. Since keys are atomic, some executions may fail when a protocol is about to output a term that contains an encryption with a non atomic key. To detect these behaviours, we need to consider slightly ill-typed traces. Formally, we consider a special constant $\omega \in \Sigma_0$. Its usefulness is illustrated in Example 4.1.2.

Definition 4.1.1. A *first-order trace* of P is a sequence $\mathsf{tr} = \mathsf{tr}_s\sigma$ where $\mathsf{tr}_s \in \mathsf{trace}_s(P)$ and σ is a substitution such that for any $\mathsf{io}(c,u)$ that occurs in tr_s with $\mathsf{io} \in \{\mathsf{in}, \mathsf{out}\}$ and u not a channel, then $u\sigma \in \mathcal{T}_0(\Sigma_c, \Sigma_0 \cup \mathcal{N} \cup \mathcal{X})$. The trace tr is said to be:

- *well-typed* w.r.t. a typing system $(\mathcal{T}, \delta)$ if there exists such a σ that is well-typed;
- *pseudo-well-typed* w.r.t. a typing system $(\mathcal{T}, \delta)$ if there exists such σ, as well as $\mathsf{c}_0 \in \Sigma_0$ and σ' such that $\sigma = \sigma'\{^{\langle\omega,\omega\rangle}/_{\mathsf{c}_0}\}$ with σ' well-typed.

Then a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ is *well-typed* (resp. *pseudo-well-typed*) if $\mathsf{tr}\phi{\downarrow}$ is well-typed (resp. *pseudo-well-typed*).

Note that Lemma 4.1.1 ensures that $\mathsf{tr}\phi\downarrow$ is a first-order trace of P, and a well-typed trace is also pseudo-well-typed.

Example 4.1.2. Going back to Example 2.4.2, let $\mathsf{tr} = \mathsf{in}(c, \langle\omega, \omega\rangle).\mathsf{out}(c, \mathsf{w}_1)$. We have that $(\mathsf{tr}, \{\mathsf{w}_1 \triangleright \mathsf{enc}(n, k)\}) \in \mathsf{trace}(P)$ while there exists no frame ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$. Consider the typing system $(\mathcal{T}, \delta)$ such that $\delta(t) = \mathsf{atom}$ for any atom or variable t and $\delta(t) = \neg\mathsf{atom}$ if t is not an atom. We can see there exists no well-typed witness of $P \not\approx Q$ (while P and Q are type-compliant as defined in Definition 2.5.3). However, the witness $(\mathsf{tr}, \{\mathsf{w}_1 \triangleright \mathsf{enc}(n, k)\})$ of $P \not\sqsubseteq Q$ is pseudo-well-typed (note that $\langle\omega, \omega\rangle$ occurs in tr). Intuitively, pseudo-well-typed traces harness the ability for the attacker to use the protocol as an oracle to test if some terms (when used in a key position) are atomic.

4.1.2 Main result

We are now ready to state our first main contribution: if there is an attack, then there is a pseudo-well-typed attack. This result holds for protocols with replications and nonces.

Theorem 4.1.1. Let P and Q be two determinate protocols type-compliant w.r.t. $(\mathcal{T}_1, \delta_1)$ and $(\mathcal{T}_2, \delta_2)$ respectively. We have that $P \not\approx Q$ if, and only if, there exists a witness of non-equivalence tr such that:

- either $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ for some ϕ and (tr, ϕ) is pseudo-well-typed w.r.t. $(\mathcal{T}_1, \delta_1)$;
- or $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ for some ψ and (tr, ψ) is pseudo-well-typed w.r.t. $(\mathcal{T}_2, \delta_2)$.

The key step for proving Theorem 4.1.1 is to provide a decision procedure, in the bounded case (*i.e.* processes without replication), that returns a pseudo-well-typed witness of non-equivalence.

Proposition 4.1.1. Let P and Q be two determinate protocols without replication. There exists an algorithm that decides whether $P \approx Q$ and if not, returns a witness tr of non-equivalence. Moreover, if P and Q are type-compliant w.r.t. $(\mathcal{T}_1, \delta_1)$ and $(\mathcal{T}_2, \delta_2)$ respectively, the witness tr of non-equivalence returned by the algorithm is such that:

- either $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ for some ϕ and (tr, ϕ) is pseudo-well-typed w.r.t. $(\mathcal{T}_1, \delta_1)$;
- or $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ for some ψ and (tr, ψ) is pseudo-well-typed w.r.t. $(\mathcal{T}_2, \delta_2)$.

The main idea is to assume given a decision procedure (for a bounded number of sessions) for reachability properties such as those proposed in [56, 34, 66] and to built on top of it a decision procedure for trace equivalence. Our procedure is carefully designed to only allow unification between encrypted subterms. To achieve this,

1. we use as a reachability blackbox one that satisfies this requirement. Most of the existing algorithms (*e.g.* [56, 34, 66]) were not designed with such a goal in mind. However, in the case of the algorithm given in [34], it has already been shown how it can be turned into one that satisfies this requirement [36].
2. we design carefully the remaining of our algorithm to only consider unification between encrypted subterms.

This design allows us to provide a pseudo-well-typed witness when the protocols under study are type-compliant and not trace equivalent.

Then, relying on Proposition 4.1.1, the proof of Theorem 4.1.1 is almost immediate. Indeed, whenever two determinate type-compliant protocols P and Q are not in trace equivalence, there exists a witness of non-inclusion for $P \sqsubseteq Q$ (or $Q \sqsubseteq P$) for a bounded version of P and Q (unfolding the replications).

4.2 A type preserving decision algorithm for bounded processes

In this section, we provide a new decision procedure for trace equivalence in the case of a bounded number of sessions to prove Proposition 4.1.1, and is thus pivotal in proving Theorem 4.1.1. The novelty of this procedure is to provide a well-typed witness whenever the two protocols are not in trace equivalence. A bounded number of sessions means formally that we consider the case of *bounded protocols*, namely determinate protocols without replication and thus without name restriction.

4.2.1 Reachability blackbox

The main idea is to assume given a decision procedure (for a bounded number of sessions) for reachability properties. Several decision procedures have already been proposed [56, 34, 66]. They are based on constraint solving techniques and even if they differ on the way the constraints are solved, the basic ideas are actually the same. These decision procedures actually do not simply say whether some state is reachable or not. They also provide a finite representation of all possible executions. More precisely, these algorithms compute a finite set of first-order (symbolic) traces that are in *solved form*, *i.e.* such that these traces are actually valid first-order traces when the variables are interpreted as constants.

Definition 4.2.1. Let $\mathsf{tr}_s = \mathsf{io}_1(c_1, u_1) \dots \mathsf{io}_n(c_n, u_n)$ be a first-order trace of P (see Definition 4.1.1). Its *associated frame* is

$$\phi_s = \{\mathsf{w}_1 \triangleright u_{i_1}, \dots, \mathsf{w}_\ell \triangleright u_{i_\ell}\}.$$

where $i_1 \dots i_\ell$ is the increasing sequence of indices that captures all the outputs of terms of the trace tr_s, *i.e.* such that $\{i_1, \dots, i_\ell\} = \{j \mid \mathsf{io}_j = \mathsf{out} \text{ and } u_j \text{ is not a channel}\}$

Definition 4.2.2. A first-order trace $\mathsf{tr}_s = io_1, \dots, io_n$ is *valid* if for all $1 \leq i \leq n$, whenever, $io_i = \mathsf{in}(c_i, u_i)$, we have that $R\phi_s\downarrow = u_i$ for some $R \in \mathcal{T}(\Sigma, \Sigma_0 \cup \mathcal{W} \cup \mathcal{X})$ where ϕ_s is the frame associated to the first-order trace $io_1 \dots io_i$ (*i.e.* tr_s up to the index i).

Executions, *i.e.* traces of $\mathsf{trace}(P)$, are exactly valid instances of symbolic traces (*i.e.*, valid instances of $\mathsf{trace}_s(P)$).

Lemma 4.2.1. Let P be a bounded protocol. We have that:

$$\{\mathsf{tr}_s\sigma \mid \mathsf{tr}_s \in \mathsf{trace}_s(P), \sigma \text{ ground and } \mathsf{tr}_s\sigma \text{ is valid}\} = \{\mathsf{tr}\phi\downarrow \mid (\mathsf{tr}, \phi) \in \mathsf{trace}(P)\}$$

Proof. The inclusion

$$\{\mathsf{tr}\phi\downarrow \mid (\mathsf{tr}, \phi) \in \mathsf{trace}(P)\} \subseteq \{\mathsf{tr}\sigma \mid \mathsf{tr}_s \in \mathsf{trace}_s(P), \sigma \text{ ground and } \mathsf{tr}_s\sigma \text{ is valid}\}$$

comes from Lemma 4.1.1 which is recalled and proven below. We need to prove that σ is ground, which can be seen in its proof, as θ is ground; and moreover $\mathsf{tr}_s\sigma$ is valid as $\mathsf{tr}\sigma = \mathsf{tr}\phi\downarrow$.

Next, we need to show that $\{\mathsf{tr}_s\sigma \mid \mathsf{tr}_s \in \mathsf{trace}_s(P), \sigma \text{ ground and } \mathsf{tr}_s\sigma \text{ is valid}\} \subseteq \{\mathsf{tr}\phi\downarrow \mid (\mathsf{tr}, \phi) \in \mathsf{trace}(P)\}$. Once again a similar induction as the one performed in the proof of Lemma 4.1.1 defines a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, the validity hypothesis ensuring that each transition in the concrete semantics is indeed possible. □

Lemma 4.1.1. Let P be a protocol and $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$. We have that $\mathsf{tr}\phi\downarrow = \mathsf{tr}_s\sigma$ for some $\mathsf{tr}_s \in \mathsf{trace}_s(P)$ and some substitution σ.

Proof. To prove this result, we consider a concrete execution $(P;\emptyset) \xrightarrow{u^1 \cdots u^n} (\mathcal{P}^n;\phi^n)$ of P and prove there exists an execution $(P;\emptyset) \xrightarrow{u_s^1 \cdots u_s^n}_s (\mathcal{P}_s^n;\phi_s^n)$ where $u_s^1 \dots u_s^n$ is a sequence of symbolic transition and such that there exists a substitution σ_n with $(\mathcal{P}^n;\phi^n) = (\mathcal{P}_s^n;\phi_s^n)\sigma_n$. To do so, we proceed by induction on $u = u^1 \dots u^n$ and first note that if u is the empty sequence, the result trivially holds.

Assuming we defined $u_s^1 \dots u_s^n$ and σ_n, we define u_s^{n+1} and σ_{n+1} as follows:

- if u^{n+1} is a τ-action (replication or name restriction), the rules, concrete and symbolic, are identical. Note that τ-transitions in this setting (without let) only depend on the structure of the protocol, ensuring that this transition is enabled in $\mathcal{P}_s^n$: we can fire $u_s^{n+1} = \tau$ (either a replication or a name restriction, like u^{n+1}). We can then define $\sigma_{n+1} = \sigma_n$ and still $(\mathcal{P}^{n+1};\phi^{n+1}) = (\mathcal{P}_s^{n+1};\phi_s^{n+1})\sigma_{n+1}$.
- if u^{n+1} is $\mathsf{out}(c, ch_i)$, $\mathcal{P}_s^n$ can fire with $u_s^{n+1} = \mathsf{out}(c, ch_i)$: then $\sigma_{n+1} = \sigma_n$, as both rules, concrete and symbolic, are identical as in the previous case. Still $(\mathcal{P}^{n+1};\phi^{n+1}) = (\mathcal{P}_s^{n+1};\phi_s^{n+1})\sigma_{n+1}$.
- if u^{n+1} is $\mathsf{in}(c, R)$, we can set $u_s^{n+1} = \mathsf{in}(c, v)$ where $\mathsf{in}(c, v)$ is the action in P which was executed as $\mathsf{in}(c, R)$ in u (which can be fired, as $\mathcal{P}_s^n\sigma_n = \mathcal{P}^n$ implies there exists such a process $\mathsf{in}(c, v).Q$ in $\mathcal{P}_s^n$ for some process Q): there exists a substitution θ such that $R\phi\downarrow = v\sigma_n\theta$ as $\mathcal{P}^n = \mathcal{P}_s^n\sigma_n$ and $u_1 \dots u_n$ is a valid execution of P. Then $\sigma_{n+1} = \sigma_n \cup \theta$ and $\mathcal{P}^{n+1} = \mathcal{P}_s^{n+1}\sigma_{n+1}$. As the frames are left unchanged, $\phi^{n+1} = \phi_s^{n+1}\sigma_{n+1}$. Note that because variables are always assumed to be independently renamed, if v binds a variable x for the first time in P, $x \notin dom(\sigma_n)$. We moreover have that $R\phi\downarrow = v\sigma_{n+1}$.
- if u^{n+1} is $\mathsf{out}(c, \mathsf{w})$, we can set $u_s^{n+1} = \mathsf{out}(c, v)$ where $\mathsf{out}(c, v)$ is the action in P which was executed as $\mathsf{out}(c, \mathsf{w})$ in u (which can be fired, as $\mathcal{P}_s^n\sigma_n = \mathcal{P}^n$ implies there exists such a process $\mathsf{out}(c, v).Q$ in $\mathcal{P}_s^n$ for some process Q): then $\sigma_{n+1} = \sigma_n$. $\mathsf{w}\phi^n = v\sigma_n$ is a consequence of $\mathcal{P}^n = \mathcal{P}_s^n\sigma_n$; which leads to $\phi^n \uplus \{\mathsf{w} \triangleright v\sigma_{n+1}\} = (\phi_s^n \uplus \{\mathsf{w} \triangleright v\})\sigma_{n+1}$ and $\mathcal{P}^{n+1} = \mathcal{P}_s^{n+1}\sigma_{n+1}$.

The result then holds by considering only observable action in u and u_s. □

Definition 4.2.3. An algorithm $\mathcal{B}$ is a *reachability blackbox* if it takes as input a first-order trace tr (issued from a bounded protocol P), and returns as output a finite set of substitutions $\sigma_1, ..., \sigma_n$ (with $dom(\sigma_i) \subseteq vars(\mathsf{tr})$) such that:

- for each i, the first-order trace $\mathsf{tr}\sigma_i$ is valid; and
- if σ is such that $\mathsf{tr}\sigma$ is a valid first-order trace of P then there exists i, and a substitution τ such that *(i)* $\mathsf{tr}\sigma = \mathsf{tr}\sigma_i\tau$, and *(ii)* for every $x \in vars(\mathsf{tr}\sigma_i)$ there exists $R_x \in \mathcal{T}(\Sigma, \Sigma_0 \cup \{\mathsf{w}_1, ..., \mathsf{w}_{\mathsf{ind}_x}\})$ such that $R_x\phi\downarrow = x\tau$ where ϕ is the frame associated to $\mathsf{tr}\sigma$ and ind_x is the number of outputs that occur in $\mathsf{tr}_s\sigma_i$ before the first occurrence of an input that contains the variable x.

All the three decision procedures proposed in [56, 34, 66] are actually reachability blackboxes.

4.2.2 Our algorithm for trace equivalence

Our algorithm $\mathcal{A}_\mathcal{B}$ makes use of a reachability blackbox $\mathcal{B}$. It takes as input two bounded protocols P and Q and returns *yes* when $P \approx Q$; and a minimal (in term of number of actions) witness tr of non-equivalence otherwise.

Our algorithm $\mathcal{A}_\mathcal{B}(P, Q)$ It consists of the following steps starting at level 1 until ℓ where ℓ denotes the maximal length (*i.e.* number of actions) of a trace in $\mathsf{trace}_s(P)$ or $\mathsf{trace}_s(Q)$. Note that since P and Q are bounded, $\mathsf{trace}_s(P)$ and $\mathsf{trace}_s(Q)$ are finite. If nothing has been returned yet (*i.e.* when the iteration steps for level ℓ has been done), then it returns *yes*, *i.e.* P and Q are trace equivalent.

Iteration steps for level n:

1. Consider every symbolic trace tr_0 in $\mathsf{trace}_s(P)$ of length n and apply $\mathcal{B}$ to it. Consider any substitution σ_0 returned by $\mathcal{B}$. We have that $\mathsf{tr}_1 = \mathsf{tr}_0\sigma_0$ is a valid first-order trace.

2. For any $s, t \in ESt(\mathsf{tr}_1)$ that are unifiable and such that $\mathsf{tr}_1\sigma_1$ is a first-order trace of P where $\sigma_1 = mgu(s, t)$, apply $\mathcal{B}$ to $\mathsf{tr}_2 = \mathsf{tr}_1\sigma_1$. Consider any substitution σ_2 returned by $\mathcal{B}$: $\mathsf{tr}_2\sigma_2$ is a valid first-order trace.

3. Consider a bijective renaming ρ from $vars(\mathsf{tr}_2\sigma_2)$ towards "fresh" public constants. Build a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ such that $\mathsf{tr}\phi\downarrow = (\mathsf{tr}_2\sigma_2)\rho$. Its existence is ensured by Lemma 4.2.1

4. Check whether there exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$. If such a frame does not exist, then return tr. Otherwise, let ψ be a resulting frame.

5. Let K_ϕ (resp. K_ψ) be the subset of $img(\rho)$ of constants occurring in key position in ϕ (resp. ψ). Check whether $K_\psi \subseteq K_\phi$. If there exists $\mathsf{c}_0 \in K_\psi \setminus K_\phi$ then return $\mathsf{tr}\{^{\langle\omega,\omega\rangle}/_{\mathsf{c}_0}\}$. Otherwise, perform step 6.

6. Check whether $\phi \sim \psi$. If the frames are not in static equivalence then return tr. Otherwise, perform steps 1 to 6 by swapping the role of P and Q.

4.2.3 Termination, soundness, and completeness

Deducibility and static equivalence are well known to be decidable for standard primitives.These two decidability results can easily be adapted in our setting. It is therefore easy to establish termination.

Proposition 4.2.1 (termination)**.** Let P and Q be two bounded protocols. The algorithm $\mathcal{A}_\mathcal{B}$ applied on P and Q terminates.

Proof. Termination is ensured by the termination of the blackbox and the decidability of static equivalence. □

A trace returned by our algorithm is indeed a witness of non-equivalence.

Proposition 4.2.2 (soundness)**.** Let P and Q be two bounded protocols. If the algorithm $\mathcal{A}_\mathcal{B}$ applied on P and Q returns a witness tr of non-equivalence, then we have that $P \not\approx Q$.

Proof. Step 4 clearly returns a witness of non equivalence. At step 5, $\mathcal{A}_\mathcal{B}$ returns a trace that is executable in P but which fails in Q since some key becomes non atomic, which yields again a witness of non equivalence. For step 6, note that checking static equivalence for only one resulting frame ψ is actually sufficient thanks to the determinacy hypothesis. □

Establishing completeness is more involved. The main difficulty is to ensure that unification performed at step 2 of the algorithm is sufficient to produce all possible relevant equalities. In particular, to capture static equivalence, we have to ensure that this is sufficient to consider tests R, R' that reduce to some encrypted subterms. The fact that we consider only unification between encrypted subterms is a key element for proving that our algorithm indeed returns a well-typed witness when P and Q are non-equivalent (cf. Section 4.2.4).

Proposition 4.2.3 (completeness)**.** Let P and Q be two bounded protocols such that $P \not\approx Q$. The algorithm $\mathcal{A}_\mathcal{B}$ applied on P and Q returns a minimal (in term of number of actions) witness tr of non-equivalence.

Proof. (Sketch) Since protocols are determinate, it is sufficient to check the alternative definition given in Definition 2.4.4 inclusion instead of static equivalence. Static inclusion, denoted $\phi \sqsubseteq_s \psi$, is when ψ satisfies all the equalities of ϕ, and $R\psi\downarrow$ is a message as soon as $R\phi\downarrow$ is a message. So if $P \not\approx Q$, there exists a witness trace tr such that $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ for some ϕ and

1. either $(\mathsf{tr}, \psi) \notin \mathsf{trace}(Q)$ for any ψ;

2. or for every ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$, we have that $\phi \not\sqsubseteq_s \psi$.

(or the contrary swapping the role of P and Q.)

In the first case, using Lemma 4.2.1, the procedure $\mathcal{B}$ would output a valid trace tr' such that $\mathsf{tr}\phi\downarrow = \mathsf{tr}'\sigma$ for some σ. We can then play tr' in Q and show that if it were a valid trace in Q, tr would also be a valid trace in Q, contradiction. We deduce that $\mathcal{A}_\mathcal{B}$ would output tr' (at step 4 or 5), a witness of non-equivalence.

In the second case, following the notation of the previous case, we have that $(\mathsf{tr}', \psi') \in \mathsf{trace}(Q)$ for some ψ' (the choice of the frame is not relevant since they are all in static equivalence due to determinacy of Q). The proof then involves a fine analysis of the relevant equalities that may yield to non static equivalence. We show that whenever there is a witness of non static inclusion for tr (this witness can be an equality test or a test checking whether a given recipe yields a message or not), then there is indeed a trace tr considered at step 2 for which we can exhibit a transformed test that witnesses non static inclusion for tr'. □

The full proof is provided in Section A.1.2.

4.2.4 Type-preservation

The specificity of the algorithm we proposed in the previous section is that it further provides a pseudo-well-typed witness whenever the two processes are not in trace equivalence. This can not be achieved using any arbitrary blackbox $\mathcal{B}$. We have to require that the blackbox $\mathcal{B}$ is *type-preserving*.

Definition 4.2.4. A reachability blackbox $\mathcal{B}$ is *type-preserving* if: for any typing system $(\mathcal{T}, \delta)$, for any protocol P type-compliant w.r.t. $(\mathcal{T}, \delta)$, for any well-typed first-order trace tr_s of P given as input, it outputs well-typed substitutions $\sigma_1, \ldots, \sigma_n$ such that:

$$ESt(\mathsf{tr}_s\sigma_i) \subseteq ESt(\mathsf{tr}_s)\sigma_i \text{ for any } i \in \{1, \ldots, n\}.$$

Lemma 4.2.2. A type-preserving reachability blackbox exists.

Most of the existing algorithms (*e.g.* [56, 34, 66]) are actually not type-preserving (since they were not designed with such a goal in mind). However, in the case of the algorithm given in [34], it has already been shown how it can be turned into a type-preserving reachability blackbox [36].

Theorem 4.2.1. Let P and Q be two bounded protocols type-compliant w.r.t. $(\mathcal{T}_1, \delta_1)$ and $(\mathcal{T}_2, \delta_2)$ respectively, and such that $P \not\approx Q$. Assume the algorithm $\mathcal{A}_\mathcal{B}$ uses a type-preserving reachability blackbox $\mathcal{B}$ and a well-typed renaming ρ at step 3. Then $\mathcal{A}_\mathcal{B}(P, Q)$ returns a trace tr such that

- either $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ for some ϕ and (tr, ϕ) is pseudo-well-typed w.r.t. $(\mathcal{T}_1, \delta_1)$;
- or $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ for some ψ and (tr, ψ) is pseudo-well-typed w.r.t. $(\mathcal{T}_2, \delta_2)$.

The proof of Theorem 4.2.1 follows from the fact that $\mathcal{A}_\mathcal{B}(P, Q)$ only manipulates well-typed traces. Indeed, it starts from traces provided by $\mathcal{B}$, which are well-typed since $\mathcal{B}$ is type-preserving. Then $\mathcal{A}_\mathcal{B}$ considers only unification between encrypted subterms (which are instances of the encrypted subterms of the protocols). Since P are Q are type-compliant, the resulting traces are well-typed. Actually, $\mathcal{A}_\mathcal{B}$ will output a well-typed trace when a failure occurs at step 4 or at step 6, and a pseudo-well-typed trace when failure occurs at step 5. The complete proof is provided in Section A.2.

4.3 Conclusion

We have proposed a general typing result that reduces the search space for attacks. For simplicity, we proved this typing result for the case of symmetric encryption and concatenation but we believe that our result could be extended to the other standard cryptographic primitives such asymmetric encryption, signatures and hash functions.

Our main typing result relies on the design of a new procedure in the case of a bounded number of sessions, that preserves typing. Specifically, we show that it is sufficient to consider only unification between encrypted (sub)terms. We think that this result can be applied to existing decision procedures (in particular SPEC [65] and also APTE [22], with some more work) to speed up their corresponding tools. As future work, we plan to implement this optimisation and measure its benefit.

Our typing result finally lays the foundation for new decidability results for trace equivalence, discussed in Chapters 5 and 6.

Part II

Decidables classes

Chapter 5

Decidability of trace equivalence for simple protocols without nonces

In this chapter, we investigate a new class of protocols for which trace equivalence is decidable for an unbounded number of sessions. Decidability results for unbounded nonces are rare and complex, even in the reachability case. One of them has been established by Ramanujam and Suresh [60] for reachability, assuming a particular tagging scheme (which itself involves nonces). More recently, Fröschle [47] proposed a procedure to decide the leakiness property, a kind of reachability property, for a tagged protocols too. Chapter 4 introduced a notion typing system as well as a typing result aiming at greatly reducing the search space when trying to check trace equivalence. The finer the typing system is, the more our typing result restricts the attack search. But in general, our typing result does not yield directly a decidability result since even the simple property of reachability is undecidable for an unbounded number of sessions and arbitrary nonces, even if the messages are of bounded size (*e.g.* [4]). Indeed, our typing system ensures the existence of a well-typed attack (if any) but the number of well-typed traces may remain infinite. To obtain decidability, we further assume a finite number of terms of each type (*i.e.* in particular a finite number of nonces). Decidability of trace equivalence for an unbounded number of sessions then follows from our main typing result, for a class of *simple* protocols where each subprocess uses a distinct channel (intuitively, a session identifiers) and no nonce generation. These restrictions indeed allow us to bound the number of messages of any type, given a *finite* typing system, which leads to a bound on the length of a minimal witness of non-equivalence (if it exists) when combined with the typing result from Chapter 4. As a corollary, we also prove decidability of trace equivalence for simple tagged protocols without nonces.

5.1 Decidability result

We consider the formal model defined in Chapter 2 with a restricted signature. We consider the signature

$$\Sigma = \{\mathsf{senc},\ \mathsf{sdec},\ \langle\ \rangle,\ \mathsf{proj}_1, \mathsf{proj}_2\} \uplus \Sigma_0$$

where Σ_0 is a set of atomic data, representing pairing and symmetric encryption as defined in Example 2.1.1. We also do not use evaluation in our processes, *i.e.* no let, and instead rely only on input filtering to model decryption operations. Moreover, we will consider only encryption with atomic keys, meaning that the set $\mathcal{M}$ of messages will be defined by $\mathcal{M} = \mathcal{M}_{\mathsf{atomic}}$.

Now, assuming finitely many terms of each type, and in particular finitely many nonces, we obtain a new decidability result for trace equivalence, for an unbounded number of sessions.

5.1.1 Main result

Our decidability result relies on the assumption that there are finitely many terms of each type (of the protocol), once the number of constants is bound for each type.

Formally, we say that a typing system $(\mathcal{T}, \delta)$ is *finite* if, for any set $\mathsf{A} \subseteq \mathcal{N} \cup \Sigma_0$ such that there is a finite number of names/constants of each type, then there are finitely many terms of each type, that is, for any $\tau \in \mathcal{T}$, the following set is finite and computable:

$$\{t \in \mathcal{T}(\Sigma_{\mathsf{c}}, \mathsf{A}) \ \mid \ \delta(t) = \tau\}.$$

Structure-preserving typing systems, as described in Definition 2.5.2 as well as the typing systems derived from tagged protocols (see Definition 2.5.6) are examples of finite typing systems.

Then, in that section, we restrict our attention to simple protocols *without nonces*, meaning protocols containing no instruction $\mathsf{new}\ n$ for any name n. Even though this limitation appears unrealistic, it is actually mitigated by the soundness result from Chapter 3, and will be later improved in Chapter 6. Note that channels names are not concerned by this restriction and instructions of the form $\mathsf{new}\ c'.\mathsf{out}(c, c')$ are still used in these protocols.

Our main theorem states that trace equivalence for simple protocols without nonces and type-compliant w.r.t. some finite typing systems is actually decidable.

Theorem 5.1.1. The problem of deciding whether two simple protocols without nonces P and Q, type-compliant w.r.t. some finite typing systems $(\mathcal{T}_1, \delta_1)$ and $(\mathcal{T}_2, \delta_2)$ are trace equivalent (*i.e.* $P \approx Q$) is decidable.

Proof. (Sketch) Since simple protocols are determinate (see Lemma 2.4.1), we obtain, thanks to our typing result (Theorem 4.1.1), the existence of well-typed witness of non-equivalence when such a witness exists. We further show that we can bound the number of useful constants in the witness trace (Proposition 5.3.1). We then derive from the finiteness of the typing system that the witness trace uses finitely many distinct terms. Therefore any trace ultimately reproduces already existing transitions. Using the form of simple protocols, we can then show how to shorten the length of the witness trace. □

We deduce then decidability of trace equivalence for simple tagged protocols without nonces.

Corollary 5.1.1. The problem of deciding whether two simple and tagged protocols without nonces P and Q are trace equivalent (*i.e.* $P \approx Q$) is decidable.

Proof. (Sketch) The first step of the proof consists in associating to a tagged protocol P, a typing system $(\mathcal{T}_P, \delta_P)$ such that P is type-compliant w.r.t. $(\mathcal{T}_P, \delta_P)$. Intuitively, $(\mathcal{T}_P, \delta_P)$ is simply induced by σ_P, the substitution ensuring the tagged condition in Definition 2.5.4. For example, the type of a closed term t is t itself while the type of a variable x in P is simply $x\sigma_P$. This definition is then propagated to any term. With such typing systems, we can show that the size of a term (*i.e.* number of function symbols) is smaller than the size "indicated" by its type (*i.e.* the size of the type, viewed as a term). Thus the typing system $(\mathcal{T}_P, \delta_P)$ is finite. We then conclude by applying Theorem 5.1.1. □

Example 5.1.1. Consider the protocol $\overline{P'_{\mathsf{OR}}}$ obtained from the protocol P'_{OR} defined in Example 2.5.5 by removing the instructions corresponding to a name restriction. These protocols are still strongly tagged and are now simple without nonces. Thus, our algorithm can be used to check whether these two protocols are in trace equivalence or not. This equivalence actually models a notion of strong secrecy of the key received by A. Since we have bounded the number of nonces, this equivalence does not require that the key is renewed at each session but it requires the key to be indistinguishable from a (private) name, n in our setting.

5.2 A sound procedure for simple protocols with nonces

Our result can be used as a proof technique to show that two simple protocols are in trace equivalence. In particular, we have that the transformation detailed in Chapter 3 can be applied to our class of protocols. Applying this to a simple, type-compliant protocol yields a process that belongs to our class to apply Theorem 5.1.1.

Proposition 5.2.1. Let $(\Sigma, \mathcal{R})$ be the theory given in Example 2.1.1 with $\mathcal{M}_\Sigma = \mathcal{M}_{\mathsf{atomic}}$. Let P and Q be two simple and type-compliant protocols built on $(\Sigma, \mathcal{R})$, and such that $\mathsf{Ch}(P) = \mathsf{Ch}(Q)$. Let N be the set of names that occur in P or Q.

The problem of deciding whether $\overline{P}^{\mathsf{N},c}$ and $\overline{Q}^{\mathsf{N},c}$ are in trace equivalence is decidable (for any $c \in \mathsf{Ch}(P)$).

5.3 Proof of Theorem 5.1.1

In this section, we provide a proof of Theorem 5.1.1.

The first proposition we prove aims at ensuring it is always possible, for simple protocols without nonces which are type-compliant w.r.t. some finite typing systems, to find a witness of non-equivalence which only uses finitely many elements of any type, which is necessary to bound the number of useful constants in said witness.

Proposition 5.3.1. There exists a set of constants and names with finitely many elements of any type such that if P and Q are simple protocols without nonces and type-compliant w.r.t. some finite typing systems $(\mathcal{T}_1, \delta_1)$ and $(\mathcal{T}_2, \delta_2)$ and $P \not\approx_t Q$ then there exists a pseudo-well-typed witness tr of non-equivalence which uses only those constants and names.

Proof. The proof here relies first on the fact that the blackbox algorithm we have described in Chapter 4 does not introduce constants or variables which were not initially in the protocol's specification. Then, as we need to cast any variable as a new constants in our algorithm to obtain a pseudo-well-typed witness of non-equivalence, we prove that we only need to preserve the difference between at most two constants of the same type. Then, by Lemma 2.4.1 and Theorem 4.1.1, we can conclude there exists a well-typed witness $\bar{\mathsf{tr}}_1$ of non-equivalence.

In the following, we assume that $\bar{\mathsf{tr}}_1$ has been discovered when applying the equivalence algorithm $\mathcal{A}_{\mathcal{B}}(P, Q)$ described in Section 4.2.1. The symmetric case is handled in the same fashion. Moreover, we can also assume the blackbox $\mathcal{B}$ used does not introduce new variables or constants as the procedure described [36], so that we can also assume that any constant in $\bar{\mathsf{tr}}_1\phi_0\downarrow$ (where $(\bar{\mathsf{tr}}_1, \phi_0) \in \mathsf{trace}(P)$) is either an element of $\Sigma_0^P \cup \mathcal{N}^P$ (the constants and names in P) or a constant introduced at step 3 of our algorithm. Let $A = \{\alpha_1, \ldots, \alpha_n\}$ be the set of such constants. We also consider a set of special constant $\mathcal{C} = \bigcup_{\tau \in \mathcal{T}^P} \mathcal{C}_\tau$ where $\mathcal{C}_\tau = \{c_1^\tau, c_2^\tau, c_3^\tau\}$ and $\delta_1(c_i^\tau) = \tau$. Note that $\mathcal{T}_0(\Sigma_{\mathsf{c}}, \cup\Sigma_0^P \cup \mathcal{N}^P \cup \bigcup_\tau \mathcal{C}_\tau)$ is such that there are only finitely many terms of any given type.

Claim: there exists a (total) renaming ρ from A to $\bigcup_\tau \mathcal{C}_\tau$ such that $\bar{\mathsf{tr}}_1\rho$ is a well-typed witness of $P \not\approx Q$ and for any term t of $\bar{\mathsf{tr}}_1\rho\phi_0\downarrow$, $t \in \mathcal{T}_0(\Sigma_{\mathsf{c}}, \cup\Sigma_0^P \cup \mathcal{N}^P \cup \bigcup_\tau \mathcal{C}_\tau)$; where ϕ_0 is such that $(\bar{\mathsf{tr}}_1, \phi_0) \in \mathsf{trace}(P)$.

Let ρ_0 be the special renaming such that for any i, if $\delta_1(\alpha_i) = \tau_i$, $\alpha_i\rho_0 = c_1^{\tau_i}$.
$\bar{\mathsf{tr}}_1$ being a witness of $\bar{P} \not\approx \bar{Q}$, several cases can occur:

- There exists ϕ and ψ such that $(\bar{\mathsf{tr}}_1, \phi) \in \mathsf{trace}(P)$ and $(\bar{\mathsf{tr}}_1, \psi) \in \mathsf{trace}(Q)$; $\phi \not\sim \psi$ and, for instance, there exist two recipes R_1 and R_2 such that $R_1\phi\downarrow = R_2\phi\downarrow$ but $R_1\psi\downarrow \neq R_2\psi\downarrow$ and all of them are messages. Let us examine $R_1\psi\downarrow$ and $R_2\psi\downarrow$. If the two terms do not share the same constructors, then $R_1(\psi\rho_0)\downarrow \neq R_2(\psi\rho_0)\downarrow$, but $R_1(\phi\rho_0)\downarrow = R_2(\phi\rho_0)\downarrow$ (as the collapsing of variables only add equalities to the frame). Now, if the two terms share the same constructors, there must exist a leaf position p in them such that $R_1\psi\downarrow|_p \neq R_2\psi\downarrow|_p$. Let us call t and s these terms respectively. If s or t is *not* an α_i for some i, then $s\rho_0 \neq t\rho_0$ as the constants in $\bigcup_\tau \mathcal{C}_\tau$ are fresh. As in the previous case, we get that $R_1(\psi\rho_0)\downarrow \neq R_2(\psi\rho_0)\downarrow$, but $R_1(\phi\rho_0)\downarrow = R_2(\phi\rho_0)\downarrow$.
  Else, assume $s = \alpha_1$ and $t = \alpha_2$, consider the renaming ρ such that, if for any i, $\delta_1(\alpha_i) = \tau_i$, $\alpha_1\rho = c_1^{\tau_1}$, $\alpha_2\rho = c_2^{\tau_2}$ and $\alpha_j\rho = c_3^{\tau_j}$ for any $j > 2$. Thus $s\rho \neq t\rho$ as the constants in $\bigcup_\tau \mathcal{C}_\tau$ are fresh, and finally we get that $R_1(\psi\rho)\downarrow \neq R_2(\psi\rho)\downarrow$, but $R_1(\phi\rho)\downarrow = R_2(\phi\rho)\downarrow$.

- Or there exists ϕ and ψ such that $(\bar{\mathsf{tr}}_1, \phi) \in \mathsf{trace}(P)$ and $(\bar{\mathsf{tr}}_1, \psi) \in \mathsf{trace}(Q)$; $\phi \not\sim \psi$ and there exists a minimal (in term of size) recipe R such that, for instance, $R\phi\downarrow$ is message while $R\psi\downarrow$ is not. If $R\psi\downarrow$ contains a element $\mathsf{dec}(\langle s, t\rangle)$ or $\mathsf{proj}_i(\mathsf{enc}(s,t))$, then $R(\psi\rho_0)\downarrow$ is not a message either, while $R(\phi\rho_0)\downarrow$ still is. Else, $R\psi\downarrow = \mathsf{dec}(\mathsf{enc}(u,v),w)$ for some terms u, v and w (by minimality) with $v \neq w$: as keys are atomic, v and w are atoms. As in the case with equalities, we can define ρ such that $v\rho \neq w\rho$, and thus $R(\psi\rho)\downarrow$ is not a message, while $R(\phi\rho)\downarrow$ is (as ρ only introduces new equalities).

- Or, finally, there exists ϕ such that $(\bar{\mathsf{tr}}_1, \phi) \in \mathsf{trace}(P)$ but for every ψ, $(\bar{\mathsf{tr}}_1, \psi) \notin \mathsf{trace}(Q)$ (the symmetric case is handled identically). If $\bar{\mathsf{tr}}_1$ end with an output, the renaming ρ_0 is adequate. So let us assume that the last action of $\bar{\mathsf{tr}}_1$ is $\mathsf{in}(c, R)$. Because protocols are simple, there exists at most one term u_P in the execution of $\bar{\mathsf{tr}}_1$ in P, and at most one term u_Q in the same execution in Q such that there exists $\exists\theta(R\phi\downarrow = u_P\theta)$, but $\forall\theta'(R\psi\downarrow \neq u_Q\theta')$. As u_Q may contain several occurrences of variables, we need to be careful to define a renaming ρ. If there exists a position p in u_Q which is not a leaf such that $u_Q|_p \neq R\psi\downarrow|_p$, then $R(\psi\rho_0)\downarrow$ and $u_Q\rho_0$ are not unifiable (they disagree on already present constructors). Else if there exists a position p in u_Q which is a leaf but not a variable such that $u_Q|_p \neq R\psi\downarrow|_p$: we define ρ as in the first subcase when dealing with an inequality without variable. Finally, if $mgu(R(\psi\rho_0)\downarrow, u_Q) = \bot$ we are done and can just take $\rho = \rho_0$; else, *i.e.* $mgu(R\psi\downarrow, u_Q) = \bot$ and $mgu(R(\psi\rho_0)\downarrow, u_Q) \neq \bot$, there exist two leaves with positions p_1 and p_2 in $R\psi\downarrow$ which corresponds to positions below variables in u_Q such that $R\psi\downarrow|_{p_1} \neq R\psi\downarrow|_{p_2}$ but $R(\psi\rho_0)\downarrow|_{p_1} = R(\psi\rho_0)\downarrow|_{p_2}$: thus we can assume $R\psi\downarrow|_{p_1} = \alpha_1$ and $R\psi\downarrow|_{p_2} = \alpha_2$. We can now define ρ such that, if for any i, $\delta_1(\alpha_i) = \tau_i$, $\alpha_1\rho = c_1^{\tau_1}$, $\alpha_2\rho = c_2^{\tau_2}$ and $\alpha_j\rho = c_3^{\tau_j}$ for any $j > 2$. Then $R(\psi\rho)\downarrow|_{p_1} \neq R(\psi\rho)\downarrow|_{p_2}$ and $mgu(R(\psi\rho)\downarrow, u_Q) = \bot$, while $mgu(R(\phi\rho)\downarrow, u_Q) \neq \bot$ as ρ only introduces new equalities.

In every case, $\bar{\mathsf{tr}}_1\rho$ is valid trace of $\bar{P}$, and $\bar{\mathsf{tr}}_1\rho$ is a trace of P with frame ϕ_0 which is well-typed (ρ is well-typed) and for any term t of $\bar{\mathsf{tr}}_1\rho\phi_0\downarrow$, $t \in \mathcal{T}_0(\Sigma_{\mathsf{c}}, \cup\Sigma_0^P \cup \mathcal{N}^P \cup \bigcup_\tau \mathcal{C}_\tau)$. □

We are now able to complete the proof of Theorem 5.1.1 by making sure any pseudo-well-typed witness of non-equivalence as discussed in Proposition 5.3.1 ultimately reproduces already existing transitions.

Theorem 5.1.1. The problem of deciding whether two simple protocols without nonces P and Q, type-compliant w.r.t. some finite typing systems $(\mathcal{T}_1, \delta_1)$ and $(\mathcal{T}_2, \delta_2)$ are trace equivalent (*i.e.* $P \approx Q$) is decidable.

Proof. By Proposition 5.3.1, if $P \not\approx_t Q$ there exists a well-typed witness of this non-equivalence built on a special set of terms. Let for instance $(\mathsf{tr}^0, \phi^0) \in \mathsf{trace}(P)$ be a minimal such witness and define $\mathsf{tr} = \mathsf{tr}^0_{-1}$, where tr^0_{-1} denotes tr^0 minus its last element, and ϕ the frame such that $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$. The minimality of tr^0 implies there exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ and $\phi \sim_s \psi$.

$$\begin{array}{rl} P = & !\mathsf{new}\ c'_1.\mathsf{out}(c_1, c'_1).B_1 \mid \ldots \mid !\mathsf{new}\ c'_m.\mathsf{out}(c_m, c'_m).B_m \\ & \mid B_{m+1} \mid \ldots \mid B_{m+n} \\ Q = & !\mathsf{new}\ c'_1.\mathsf{out}(c_1, c'_1).B'_1 \mid \ldots \mid !\mathsf{new}\ c'_m.\mathsf{out}(c_m, c'_m).B'_m \\ & \mid B'_{m+1} \mid \ldots \mid B'_{m+n'} \end{array}$$

(We can safely assume the same number of restricted channels (m) in P and Q, as non-equivalence would trivially hold otherwise.) We define a function proj_P which takes a channel c_i and returns $\{\mathsf{tr}|_{c_i,c'_i}\}_{c'_i}$ the set of subtraces of tr corresponding of actions on channel c'_i and the action $\mathsf{new}\ c'_i.\mathsf{out}(c_i, c'_i)$ which originated it. It can be extended naturally to channels c_{m+1} to c_{m+n} (in that case, in absence of replication, each set is a singleton, which, for the sake of uniformity, we will denote by $\{\mathsf{tr}|_{c_i,c'_i}\}_{c'_i}$ for $i \in \{m+1, \ldots, m+n\}$ too). We can similarly define proj_Q. We now claim:

Claim 1: $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q) \Rightarrow \mathsf{proj}_P = \mathsf{proj}_Q$, as subprocesses B_i and B'_i use public fresh channels such that each c'_i is spawned from a unique channel c_i which is visible in tr.

Moreover we define a relation $\mathcal{R}_P$ on elements of $img(\mathsf{proj}_P)$. We say that $\mathsf{tr}|_{c_i,c'_i} \preceq_P \mathsf{tr}|_{c_i,c''_i}$ if, and only if, $\mathsf{tr}|_{c_i,c'_i}\phi\downarrow$ is a prefix of $(\mathsf{tr}|_{c_i,c''_i}\phi\downarrow)\{^{c'_i}/_{c''_i}\}$. $\mathcal{R}_P$ is then defined as $\preceq_P \cap \preceq_P^{-1}$; and is an equivalence relation.

Symmetrically, we can define a relation $\mathcal{R}_Q$ with ψ and make the following claim:

Claim 2: $\mathcal{R}_P = \mathcal{R}_Q = \mathcal{R}$, as tr^0 is a minimal witness of $P \not\approx Q$. Indeed, if there are two elements $\mathsf{tr}|_{c_i,c_i'}$ and $\mathsf{tr}|_{c_i,c_i''}$ such that $\mathsf{tr}|_{c_i,c_i'}\ \mathcal{R}_P\ \mathsf{tr}|_{c_i,c_i''}$ but $\mathsf{tr}|_{c_i,c_i'}$ and $\mathsf{tr}|_{c_i,c_i''}$ are not $\mathcal{R}_Q$-equivalent (or the other way around), then $\mathsf{tr}|_{c_i,c_i'}\psi\downarrow$ is not a prefix of $(\mathsf{tr}|_{c_i,c_i''}\psi\downarrow)\{^{c_i'}/_{c_i''}\}$ (or, once again, the other way around). Thus there exists an action $\mathsf{a}(c_i', R)$ ($\mathsf{a} \in \{\mathsf{in}, \mathsf{out}\}$) of $\mathsf{tr}|_{c_i,c_i'}$ and an action $\mathsf{a}(c_i'', R')$ of $\mathsf{tr}|_{c_i,c_i''}$ (these actions must be two inputs or two outputs as $\mathsf{tr}|_{c_i,c_i'}\ \mathcal{R}_P\ \mathsf{tr}|_{c_i,c_i''}$) such that $R\psi\downarrow \neq R'\psi\downarrow$ while $R\phi\downarrow = R'\phi\downarrow$ (as $\mathsf{tr}|_{c_i,c_i'}\ \mathcal{R}_P\ \mathsf{tr}|_{c_i,c_i''}$), which would contradict the minimality of tr^0.

Intuitively, two subtraces are in equivalence w.r.t. the relation $\mathcal{R}$ if they are identical, modulo the name of the fresh channel they use. The general idea to then bound the length of tr is to observe that if tr contains two equivalent subtraces, then one of them, the one occurring "later" can be safely removed as it adds no new branch or new terms to the execution. That way, we will prove that because tr^0 is a minimal witness, tr can only contain one representative of each class, otherwise we could obtain a shorter witness by eliminating this extra element. To do so, we define which actions should be kept during this elimination and how to generate the new trace.

Given an action α we map it to $\beta(\alpha)$ its occurrence index in tr. Consequently we can define $\min(\alpha, \alpha')$ as the action with the lowest occurrence index; and lift it to sequences of actions:

$$\min(\alpha_1 \dots \alpha_n, \alpha_1' \dots \alpha_n') = \min(\alpha_1, \alpha_1') \dots \min(\alpha_n, \alpha_n').$$

Similarly, we can define $\max(\alpha, \alpha')$ as the action with the highest occurrence index in tr and lift it to sequence of actions.

$$\max(\alpha_1 \dots \alpha_n, \alpha_1' \dots \alpha_n') = \max(\alpha_1, \alpha_1') \dots \max(\alpha_n, \alpha_n').$$

We define an operation $\mathsf{merge}(\mathsf{tr}, \mathsf{tr}|_{c_j,c_j'}, \mathsf{tr}|_{c_j,c_j''})$ where $\mathsf{tr}|_{c_j,c_j'}\ \mathcal{R}\ \mathsf{tr}|_{c_j,c_j''}$, $\mathsf{tr}|_{c_j,c_j'} = \alpha_1 \dots \alpha_n$ and $\mathsf{tr}|_{c_j,c_j''} = \alpha_1' \dots \alpha_n'$. Note that if two subtraces are equivalent w.r.t. $\mathcal{R}$ they must have the same length as they must each be a prefix of the other. For each $i \in \{1, \dots, n\}$, we want to delete from tr the later action $\max(\alpha_i, \alpha_i')$, while renaming the channel used by the earlier action $\min(\alpha_i, \alpha_i')$ to c_j' (we assume that $\min(\alpha_1, \alpha_1') = \alpha_1 = \mathsf{new}\ c_j'.\mathsf{out}(c_j, c_j')$, without loss of generality) and rename in tr all the occurrences of deleted output variables. We denote by $\mathsf{w}_i^{\max}$ the variable used by $\max(\alpha_i, \alpha_i')$ if it is an output, *i.e.* if $\max(\alpha_i, \alpha_i') = \mathsf{out}(c, \mathsf{w}^{\max})$ for some c and w; and by $\mathsf{w}_i^{\min}$ the variable used by $\min(\alpha_i, \alpha_i')$ if it is an output, *i.e.* if $\min(\alpha_i, \alpha_i') = \mathsf{out}(c, \mathsf{w}^{\min})$ for some c and w. Because $\alpha_1 \dots \alpha_n\ \mathcal{R}\ \alpha_1' \dots \alpha_n'$, $\mathsf{w}_i^{\max}\phi\downarrow = \mathsf{w}_i^{\min}\phi\downarrow$ and $\mathsf{w}_i^{\max}\psi\downarrow = \mathsf{w}_i^{\min}\psi\downarrow$ for any $i \in \{1, \dots, n\}$. We denote by σ the substitution that maps $\mathsf{w}_i^{\max}$ to $\mathsf{w}_i^{\min}$ for any $i \in \{1, \dots, n\}$. Let then tr^- be tr minus any action in $\max(\alpha_1 \dots \alpha_n, \alpha_1' \dots \alpha_n')$: we define $\mathsf{merge}(\mathsf{tr}, \mathsf{tr}|_{c_j,c_j'}, \mathsf{tr}|_{c_j,c_j''}) = \mathsf{tr}^-\sigma\{^{c_j'}/_{c_j''}\}$.

In particular, there exist ϕ' and ψ' such that $(\mathsf{merge}(\mathsf{tr}, \mathsf{tr}|_{c_j,c_j'}, \mathsf{tr}|_{c_j,c_j''}), \phi') \in \mathsf{trace}(P)$, $(\mathsf{merge}(\mathsf{tr}, \mathsf{tr}|_{c_j,c_j'}, \mathsf{tr}|_{c_j,c_j''}), \psi') \in \mathsf{trace}(Q)$. $\mathsf{merge}(\mathsf{tr}, \mathsf{tr}|_{c_j,c_j'}, \mathsf{tr}|_{c_j,c_j''})$ is indeed a valid trace in both P and Q: the actions of $\min(\mathsf{tr}|_{c_j,c_j'}, \mathsf{tr}|_{c_j,c_j''})$ occur on the same channel c_j', as $\min(\alpha_1, \alpha_1') = \alpha_1 = \mathsf{new}\ c_j'.\mathsf{out}(c_j, c_j')$ and because $\mathsf{w}_i^{\max}$ is renamed to $\mathsf{w}_i^{\min}$ for any $i \in \{1, \dots, n\}$, for any recipe R in tr, $R\phi\downarrow = (R\sigma)\phi\downarrow$ (and $R\psi\downarrow = (R\sigma)\psi\downarrow$) and for any transition $(\mathcal{P}_k, \phi_k) \xrightarrow{\mathsf{in}(c,R)} (\mathcal{P}_{k+1}, \phi_{k+1})$ in tr, $vars(R\sigma) \subseteq dom(\phi_k)$ as we kept the first occurrence of any output with $\min(\alpha_i, \alpha_i')$. Moreover we get that $\phi' \approx \psi'$. Indeed, $\mathsf{merge}(\mathsf{tr}, \mathsf{tr}|_{c_j,c_j'}, \mathsf{tr}|_{c_j,c_j''})\phi'\downarrow$ is a subtrace of $\mathsf{tr}\{^{c_j'}/_{c_j''}\}\phi\downarrow$, and $\mathsf{merge}(\mathsf{tr}, \mathsf{tr}|_{c_j,c_j'}, \mathsf{tr}|_{c_j,c_j''})\psi'\downarrow$ is a subtrace of $\mathsf{tr}\{^{c_j'}/_{c_j''}\}\psi\downarrow$.

Let $[\mathsf{tr}_1], \dots, [\mathsf{tr}_M]$ be the equivalence classes of $\mathcal{R}$, then, by minimality of tr^0, each class has at exactly one element. If some class $[\mathsf{tr}_i]$ were to have two elements $\mathsf{tr}|_{c_j,c_j'}$ and $\mathsf{tr}|_{c_j,c_j''}$, $\mathsf{merge}(\mathsf{tr}, \mathsf{tr}|_{c_j,c_j'}, \mathsf{tr}|_{c_j,c_j'}).(\alpha\sigma\{^{c_j'}/_{c_j''}\})$ would provide a shorter witness of $P \not\approx Q$, where α is the last action of tr^0 (which was excluded in tr).

From Proposition 5.3.1, we know there exists a set of atoms such that there exists only finitely element of the same type and such that any pseudo-well-typed trace only uses those. As the type systems we consider are finite, there exists only finitely many terms of each type. And finally, as the trace is pseudo-well-typed, each type in $\mathsf{tr}^0\phi^0\downarrow$ appears in P. Hence there are only finitely types and finitely many terms that can occur in any pseudo-well-typed trace of P: let T be that number. Thus we claim that $M \leq (n+m) \times \sum_{i=0}^{B} T^i$ where B is the maximal length (in terms of number

of actions) of a parallel branch in P, *i.e.* $\max_k |\mathsf{tr}_k|$. Indeed a class is defined by its sequence of actions (bounded by the maximum number of actions in any branch of P) and the first-order terms it contains (which is bounded by the total number of existing eligible terms). As there are $n + m$ branches in P, of length at most B, there are at most $(n + m)T^i$ different first-order sequences of length $i \in [0, B]$ in P. Then, as $|\mathsf{tr}| \leq M \times B$ (tr contains at most one representative for each equivalence class, each being of length at most B), we finally get $|\mathsf{tr}| \leq (n+m) \times B \times \sum_{i=0}^{B} T^i$. Hence, an upper bound on the length of a well-typed witness of $P \not\approx_t Q$ is $N = (n+m)B\sum_{i=0}^{B} T^i + 1$, and the number of such traces in $\mathsf{trace}(P)$ is bounded by $(2T)^N$ as each element of a trace is either an input or an output of a term, which provides a straightforward algorithm to decide trace equivalence. □

5.4 Proof of Corollary 5.1.1

For every tagged protocol, we can define a induced typing system as shown in Definition 2.5.6 which is finite, assuming a bounded number of nonces, to prove Corollary 5.1.1. We recall that any tagged protocol is actually type-compliant w.r.t. its induced typing system, as per Proposition 2.5.1.

Corollary 5.1.1. The problem of deciding whether two simple and tagged protocols without nonces P and Q are trace equivalent (*i.e.* $P \approx Q$) is decidable.

Proof. Since P (resp. Q) is tagged, thanks to Proposition 2.5.1, we know that P (resp. Q) is type-compliant w.r.t. $(\mathcal{T}_P, \delta_P)$ (resp. $(\mathcal{T}_Q, \delta_Q)$), the typing system associated to P (resp. Q) as defined in Definition 2.5.6. With such typing systems, we have that the size of a term (*i.e.* number of function symbols) is smaller that the size "indicated" by its type (*i.e.* the size of the type, viewed as a term). Thus, it is then easy to see that the set:

$$\{t \in \mathcal{T}(\Sigma_{\mathsf{c}}, \mathsf{A}) \mid \delta(t) = \tau\}$$

is finite for any $\tau \in \mathcal{T}_P$ (and similarly for Q) as soon as A is a set of names and constants that contains only a finite number of names/constants of each type. We conclude by applying Theorem 5.1.1. □

5.5 Conclusion

From a general typing result that reduces the search space for attacks described in Chapter 4, we derive decidability for an unbounded number of sessions and simple protocols, assuming a finite number of nonces. This result can naturally be applied to simple tagged protocols without nonces as well. Even though limiting the number of nonces seems quite impractical, Chapter 3 offers a sound approach to lift this decidability result to a sound terminating procedure for checking trace equivalence with arbitrary nonces. Because our proof method to achieve decidability relies on bounding the number of terms occurring in a minimal witness of non-equivalence, it suggests further results could be obtained from existing decidability for equivalence in process algebra *without terms*, such as the processes described in [29].

Chapter 6

Decidability of trace equivalence for acyclic simple protocols with nonces

We propose in this chapter the first decidability result for trace equivalence, for an unbounded number of sessions and with nonces. Since even simple reachability properties are undecidable in this context, we make some assumptions.

Simple processes. This notion has been introduced in [33] and is defined in Chapter 2. Intuitively, we assume that each process communicates on a distinct channel. In practice, each machine has its own IP address and each session is characterised by some session identifier. We also assume that each process consists of a sequence of inputs and outputs (with some tests). This models very well standard security protocols (with no else branches).

Type compliant protocols. Intuitively, we assume that ciphertexts cannot be confused. A similar notion has been formally introduced in [15] and developed in Chapters 2 and 4 and was shown to ensure termination of ProVerif (without nonces). This condition is part of the good design practices and is easy to enforce by adding some identifier (a tag) in each ciphertext. Of course the same tags are re-used in all sessions.

Acyclic dependency graph. Considering constructions used in undecidability results, one can notice that the encodings rely on some form of cyclicity. Typically, the last message of the protocol is re-injected at the first step of the protocol, forming an infinite loop. We therefore introduce the notion of dependency graph, with two notions of dependencies:

- sequential dependency: some action can only be taken after some other actions;
- data dependency: some message can only be built once some information is learnt from another message.

This graph can be computed (automatically) from the protocol's specification. To detect data dependencies, we actually consider a particular typed instantiation of the protocol. Therefore, the definition of a dependency graph relies itself on the typing system. Moreover, finer typing systems are more likely to yield acyclic dependency graphs.

The objective of this chapter is to show that the equivalence between simple and type-compliant protocols with an acyclic dependency graph is decidable, for protocols using symmetric encryption, concatenation, and nonces. Our class encompasses most symmetric key protocols we considered, including Needham-Schroeder with symmetric key, Otway-Rees, Denning-Sacco, or Wide-Mouthed-Frog. For some of these protocols, we had to consider an explicitly tagged version.

Proof technique. We show decidability in two main steps. First, thanks to the type-compliance assumption, we show that we can apply the typing result from Chapter 4, yielding a bound on the size of the messages: if there is a witness of non-equivalence then there is a well-typed witness, and this induces a strict format for the messages occurring in such a witness. Note that the number of distinct messages remains unbounded due to nonces.

The second step of the proof relies on the dependency graph. We show that any well-typed execution trace complies with the execution order induced by the dependency graph, which allows to split well-typed traces into small independent traces, which in turn yields decidability.

Scope. The scope of our result depends on how often protocols induce an acyclic dependency graph. For the sake of clarity, we first provide a generic definition of a *dependency graph* (Definition 6.2.2). However, some interesting protocols such as the Needham-Schroeder symmetric key protocol are cyclic with this definition. In a second step, we provide a criterion that safely allows to remove edges in the dependency graph, yielding acyclicity for most of the protocols we considered. This more flexible notion of dependency graph is called *refined dependency graph* (Definition 6.3.4). Contemporaneously to our work, Sybille Fröschle [47] has proposed a new decidability result for the "leakiness" property and the class of "well-founded protocols". We provide in Section 6.4.3 a detailed comparison with our result. In brief, our result are incomparable since [47] considers a larger class of primitives but a less accurate security property and more restriction on the protocols (*e.g.* ciphertext forwarding is again prohibited and as in [54] a typed model is considered). We believe that our approach provides a good level of flexibility. In case some protocols were found to be cyclic with our current definition of a dependency graph, it should be possible to develop other criteria that soundly remove edges.

6.1 Annotated model for security protocols

In addition to the model we already introduced in Chapter 2 and precised in Chapter 4, we assume annotations in our protocols. These annotations are meant to make it easier to pinpoint a particular action in the protocol specification and are completely invisible to the attacker.

More precisely, we assume an infinite set $\mathcal{L}$ used to name input and output actions of processes. Protocols are now modelled through processes built by the following grammar:

$$\begin{array}{rcl} P, Q & := & 0 \\ & | & \alpha : \mathsf{in}(c, u).P \\ & | & \alpha : \mathsf{out}(c, u).P \\ & | & (P \mid Q) \\ & | & !P \\ & | & \mathsf{new}\ n.P \\ & | & \mathsf{new}\ c'.\mathsf{out}(c, c').P \end{array}$$

where $u \in \mathcal{T}(\Sigma_{\mathsf{c}}, \Sigma_0 \cup \mathcal{N} \cup \mathcal{X})$, $n \in \mathcal{N}$, $c, c' \in \mathcal{C}h$, and $\alpha \in \mathcal{L}$. The only difference with the process algebra defined in Section 2.1 is the presence of a label α in $\alpha : \mathsf{in}(c, u).P$ and $\alpha : \mathsf{out}(c, u).P$, to serve as an easy way to later refer to these actions.

Example 6.1.1. The Denning Sacco protocol described in Example 2.2.1 can be rewritten with labels as follows. k_{as}, k_{bs}, k_{ab} are names, whereas a and b are constants from Σ_0. The protocol is modelled by the parallel composition of three processes P_A, P_B, and P_S, corresponding to the roles of A, B, and S.

$$\begin{array}{rl} P_{\mathsf{DS}} = & !\,\mathsf{new}\ c_1.\mathsf{out}(c_A, c_1).P_A \mid !\,\mathsf{new}\ c_2.\mathsf{out}(c_B, c_2).P_B \\ & \mid !\,\mathsf{new}\ c_3.\mathsf{out}(c_S, c_3).P_S \end{array}$$

The processes P_A, P_B, and P_S are given below.

$$\begin{array}{rcl} P_A & = & \alpha_1 : \mathsf{out}(c_1, \langle \mathsf{a}, \mathsf{b} \rangle). \\ & & \alpha_2 : \mathsf{in}(c_1, \mathsf{enc}(\langle \mathsf{b}, x_{AB}, x_B \rangle, k_{as})). \\ & & \alpha_3 : \mathsf{out}(c_1, x_B) \\ P_B & = & \beta_1 : \mathsf{in}(c_2, \mathsf{enc}(\langle y_{AB}, \mathsf{a} \rangle, k_{bs})) \\ P_S & = & \gamma_1 : \mathsf{in}(c_3, \langle \mathsf{a}, \mathsf{b} \rangle).\ \mathsf{new}\ k_{ab}. \\ & & \gamma_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle \mathsf{b}, k_{ab}, \mathsf{enc}(\langle k_{ab}, \mathsf{a} \rangle, k_{bs}) \rangle, k_{as})) \end{array}$$

The semantics are left virtually unchanged by the annotations, which lets us use the same definitions as in Chapter 2. Figure 6.1 has the same transitions rules as Figure 2.1, leading to the same semantics as in Section 2.3. Note in particular that labels are absent from the transitions in Figure 6.1, which leaves them unobservable for the attacker and leads to the same set of traces as the un-annotated version of the protocol.

$$(\alpha : \mathsf{in}(c, u).P \cup \mathcal{P}; \phi) \xrightarrow{\mathsf{in}(c,R)} (P\sigma \cup \mathcal{P}; \phi) \quad \text{where } R \text{ is a recipe such that } R\phi\downarrow \text{ is a message and } R\phi\downarrow = u\sigma \text{ for some } \sigma \text{ with } dom(\sigma) = vars(u)$$

$$(\alpha : \mathsf{out}(c, u).P \cup \mathcal{P}; \phi) \xrightarrow{\mathsf{out}(c,\mathsf{w}_{i+1})} (P \cup \mathcal{P}; \phi \cup \{\mathsf{w}_{i+1} \triangleright u\}) \quad \text{where } u \text{ is a message and } i \text{ is the number of elements in } \phi$$

$$(\mathsf{new}\, c'.\mathsf{out}(c, c').P \cup \mathcal{P}; \phi) \xrightarrow{\mathsf{out}(c,ch_i)} (P\{^{ch_i}/_{c'}\} \cup \mathcal{P}; \phi) \quad \text{where } ch_i \text{ is the "next" fresh channel name available in } \mathcal{C}h^{\mathsf{fresh}}$$

$$(\mathsf{new}\, n.P \cup \mathcal{P}; \phi) \xrightarrow{\tau} (P\{^{n'}/_{n}\} \cup \mathcal{P}; \phi) \quad \text{where } n' \text{ is a fresh name in } \mathcal{N}$$

$$(!P \cup \mathcal{P}; \phi) \xrightarrow{\tau} (P \cup\, !P \cup \mathcal{P}; \phi)$$

Figure 6.1: Annotated semantics of the processes

The definition of simple processes has to be slightly adapted too, to account for the new labels.

Definition 6.1.1. A *simple protocol* P is a protocol of the form

$$\begin{array}{c} !\mathsf{new}\ c'_1.\mathsf{out}(c_1, c'_1).B_1 \mid \ldots \mid !\mathsf{new}\ c'_m.\mathsf{out}(c_m, c'_m).B_m \\ \mid B_{m+1} \mid \ldots \mid B_{m+n} \end{array}$$

where each B_i with $1 \leq i \leq m$ (resp. $m < i \leq m + n$) is a ground process on channel c'_i (resp. c_i) built using the following grammar:

$$B := 0 \mid \alpha : \mathsf{in}(c'_i, u).B \mid \alpha : \mathsf{out}(c'_i, u).B \mid \mathsf{new}\, n.B$$

where $u \in \mathcal{T}_0(\Sigma_{\mathsf{c}}, \Sigma_0 \cup \mathcal{N} \cup \mathcal{X})$. Moreover, we assume that $c_1, \ldots, c_n, c_{n+1}, \ldots, c_{n+m}$ are pairwise distinct, as well as the labels for each action.

Given a simple protocol P, and $\alpha, \beta \in \mathcal{L}(P)$, we say that β *directly follows* α in P if both actions are in sequence in the description of P, with β after α, and no other visible action in between. When some other visible actions occur between α and β, we simply say that β *follows* α.

6.2 A first decidability result

Trace equivalence is undecidable in general for an unbounded number of sessions, inheriting undecidability from the standard secrecy case (see *e.g.* [44]). We present here our two main assumptions for obtaining decidability: *type-compliance* and *acyclic dependency graph*. For the clarity of the presentation, we present in this section a rather coarse definition of dependency graph. In the next section, we provide a sound criterion to remove some of its edges.

6.2.1 Dependency graph

We consider a protocol P which is type-compliant w.r.t. a structure-preserving typing system $(\mathcal{T}_P, \delta_P)$. To define our dependency graph, we first define public and honest types. Terms of public type are always deducible for an adversary, whereas terms of honest type will be guaranteed to be secret (except those built using public constants only).

A type τ_p is *public* if $\delta_P(n) \notin St(\tau_p)$ for any name n occurring in P. Intuitively, in a well-typed execution, a term having a public type is a term built using public constant only, and is thus deducible from the beginning of any execution.

An atomic type τ_h is *honest* if *(i)* τ_h does not appear in plaintext position in $u\delta_P$ for any term u occurring in P; *(ii)* $\tau_h \neq \delta_P(a)$ for any constant/variable a occurring in P. Intuitively, this ensures that, in a well-typed execution, terms of type τ_h will never occur in plaintext position, and no public constant of this type will be used in key position.

Example 6.2.1. Going back to Example 6.1.1, we have that τ_{a}, τ_{b}, τ_{m} are public types while τ_{kas} and τ_{kbs} are honest types. In contrast, τ_{kab} is neither a public nor an honest type. Indeed, $\tau_{kab} = \delta_P(y_{AB})$ and the variable y_{AB} occurs in P.

We define inductively a function ρ_{io} that inspects a type τ and returns its set of deducible subterms (where τ is viewed as a term) together with the set of keys needed to access each subterm. For this, we introduce a new syntactic symbol $\#$.

Definition 6.2.1. Given a type τ, a position p and a set S of types, the function ρ_{io} is inductively defined as follows:

- $\rho_{\mathsf{io}}(\tau_0, p, S) = \{(\tau_0, p)\#S\}$ for any atomic type τ_0;
- $\rho_{\mathsf{io}}(\langle\tau_1, \tau_2\rangle, p, S) = \rho_{\mathsf{io}}(\tau_1, p.1, S) \cup \rho_{\mathsf{io}}(\tau_2, p.2, S)$;
- $\rho_{\mathsf{io}}(\mathsf{enc}(\tau_1, \tau_2), p, S) =$
  $\begin{cases} \{(\mathsf{enc}(\tau_1, \tau_2), p)\#S\} \text{ if } \tau_2 \text{ is an honest type;} \\ \{(\mathsf{enc}(\tau_1, \tau_2), p)\#S\} \cup \rho_{\mathsf{io}}(\tau_1, p.1, S \cup \{\tau_2\}) \text{ otherwise.} \end{cases}$

Given a type τ, the function $\rho_{\mathsf{io}}(\tau, \epsilon, \emptyset)$ computes a set of elements of the form $(\tau_i, p_i)\#S_i$. Intuitively, it means that the term of type τ_i at position p_i in τ is accessible from the term τ after some decryptions using keys occurring in the set S_i.

We also define two functions ρ_{out} and ρ_{in} that help us to define the flows that may happen during a protocol execution.

$$\rho_{\mathsf{out}}(\tau') = \{(\tau, p) \mid (\tau, p)\#S \in \rho_{\mathsf{io}}(\tau', \epsilon, \emptyset)\}; \text{ and}$$
$$\rho_{\mathsf{in}}(\tau') = \{\tau, \tau_1, \ldots, \tau_k \mid (\tau, p)\#\{\tau_1, \ldots, \tau_k\} \in \rho_{\mathsf{io}}(\tau', \epsilon, \emptyset)\}.$$

Intuitively, $\rho_{\mathsf{out}}(\tau')$ returns the types of the terms that may be deducible by the attacker once a term of type τ' is outputted, whereas $\rho_{\mathsf{in}}(\tau')$ returns all the types that may be used by the attacker to fill an input of type τ'. In case of an output, we also return the position at which the type occurred. This information will be added in our dependency graph, and used in Section 6.3 to present our refined dependency graph.

Example 6.2.2. Continuing our running example, we have that:
$\rho_{\mathsf{out}}(\langle\tau_{\mathsf{a}}, \tau_{\mathsf{b}}\rangle) = \{(\tau_{\mathsf{a}}, 1), (\tau_{\mathsf{b}}, 2)\}$ and $\rho_{\mathsf{in}}(\langle\tau_{\mathsf{a}}, \tau_{\mathsf{b}}\rangle) = \{\tau_{\mathsf{a}}, \tau_{\mathsf{b}}\}$;
$\rho_{\mathsf{out}}(\mathsf{enc}(\langle y_{AB}\delta_P, \tau_{\mathsf{a}}\rangle, \tau_{kbs})) = \{(\mathsf{enc}(\langle\tau_{kab}, \tau_{\mathsf{a}}\rangle, \tau_{kbs}), \epsilon)\}$;
$\rho_{\mathsf{in}}(\mathsf{enc}(\langle y_{AB}\delta_P, \tau_{\mathsf{a}}\rangle, \tau_{kbs})) = \{\mathsf{enc}(\langle\tau_{kab}, \tau_{\mathsf{a}}\rangle, \tau_{kbs})\}$.
since τ_{kbs} is an honest type.

We are now ready to define the dependency graph. It captures two main sources of dependencies: sequential dependencies, when an action may only occur after another one, and data dependencies, when the production of a term depends on other sent terms.

Definition 6.2.2. The *dependency graph* associated to a type-compliant, simple protocol P (w.r.t. a structure-preserving typing system $(\mathcal{T}_P, \delta_P)$) is a graph having $\mathcal{L}(P)$ as vertices and that are connected as follows:

1. for every action with label α in P that directly follows an action with label β in P, there is an edge $\alpha \to \beta$;
2. for every "$\alpha : \mathsf{in}(c, u)$" and "$\beta : \mathsf{out}(d, v)$" in P, there is an edge $\alpha \to^p \beta$ if there exists $\tau \in \rho_{\mathsf{in}}(u\delta_P)$ such that
$$(\tau, p) \in \rho_{\mathsf{out}}(v\delta_P)$$
and τ is not a public type.
3. for every "$\alpha : \mathsf{out}(c, u)$" and "$\beta : \mathsf{out}(d, v)$" in P, there is an edge $\alpha \to^p \beta$ if $(\tau, q)\#(S \cup \{\tau_k\}) \in \rho_{\mathsf{io}}(u\delta_P, \epsilon, \emptyset)$ for some τ, q, S, and τ_k such that
$$(\tau_k, p) \in \rho_{\mathsf{out}}(v\delta_P)$$
and τ_k is not a public type.

Intuitively, if there exists an edge from α to β, it implies that the action α *may* depend on action β. More precisely, item 1 captures sequential dependencies, whereas items 2 and 3 are about data dependencies. Item 2 captures dependencies that occur due to the fact that the attacker need to produce a term to comply with the given input. For this, all the needed pieces may come from different outputs (but at a plaintext position). Now, item 3 is needed because, when such a piece occurs at a plaintext position (for instance under an encryption with k), it may be important for the attacker to learn the key k, and this generates new dependencies.

Example 6.2.3. The dependency graph for the protocol P^1_{DS} defined in Example 2.4.3 w.r.t. the typing system $(\mathcal{T}_{\mathsf{DS}}, \delta_{\mathsf{DS}})$ given in Example 2.5.2 is depicted in Figure 6.2. The vertical arrows correspond to sequential dependencies (item 1) whereas all the other arrows are actually due to item 2. In this example, item 3 does not produce any arrow.

Intuitively, these arrows mean that the input α_2 (resp. β_1) may depend on the output γ_2 (resp. α_3). In other words the outputted term may be (partially) used to fill the input. The relevant parts of the output are indicated by the position p on top of the arrows.

Note that for P^2_{DS} (also defined in Example 2.4.3), we can introduce an additional atomic type τ_k for name k (or reuse the atomic type τ_{kab}. In both cases, the dependency graph of P^2_{DS} will be exactly the same as the one obtained for P^1_{DS}.

Example 6.2.4. Let P be the protocol

$$P = \alpha : \mathsf{in}(c_1, k_1) | \, \beta : \mathsf{out}(c_2, \mathsf{enc}(k_1, k_2)) | \, \gamma : \mathsf{out}(c_3, k_2)$$

Consider the typing system $(\mathcal{T}, \delta)$ where $\mathcal{T} = \{\tau_1, \tau_2\}$; $\delta(k_1) = \tau_1$ and $\delta(k_2) = \tau_2$. Its dependency graph is shown in Figure 6.3. The edge from β to γ is an edge introduced by the third item in Definition 6.2.2. Indeed, $\rho_{\mathsf{io}}(\mathsf{enc}(\tau_1, \tau_2), \epsilon, \emptyset) = \{(\mathsf{enc}(\tau_1, \tau_2), \epsilon)\#\emptyset, (\tau_1, 1)\#\{\tau_2\}\}$ and $\rho_{\mathsf{out}}(\tau_2) = \{(\tau_2, \epsilon)\}$.

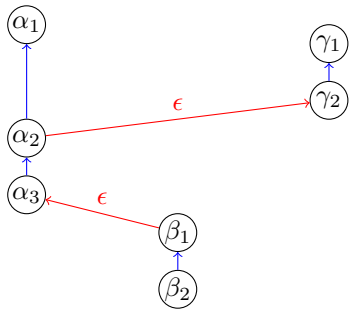


Figure 6.2: Dependency graph for P^1_{DS} w.r.t. $(\mathcal{T}_{\mathsf{DS}}, \delta_{\mathsf{DS}})$

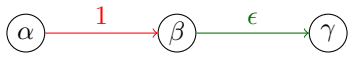


Figure 6.3: Dependency graph for P w.r.t. $(\mathcal{T}, \delta)$

6.2.2 Our result

Trace equivalence is decidable for simple, type-compliant, acyclic protocols.

Theorem 6.2.1. Let P and Q be two simple protocols type-compliant w.r.t. some structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$, and with acyclic dependency graphs. The problem of deciding whether P and Q are in trace equivalence (*i.e.* $P \approx Q$) is decidable.

Theorem 6.2.1 will be proved in Section 6.5 in a more general setting.

Example 6.2.5. The protocols P^1_{DS} and P^2_{DS} given in Example 2.4.3 are simple, type-compliant w.r.t. $(\mathcal{T}_{\mathsf{DS}}, \delta_{\mathsf{DS}})$ and their respective dependency graph is acyclic. Thus these protocols fall into our decidable class.

6.3 An improved version of our decidability result

In the previous section we have presented a first decidability result for trace equivalence of simple, type-compliant, acyclic protocols. The Denning-Sacco protocol satisfies these hypotheses. However, some reasonable protocols do not fall in our class. In the next paragraph, we explain why the Needham-Schroeder protocol induces a cyclic dependency graph. However, in that case, the cycle is created by a false dependency. Therefore, in the subsequent paragraphs, we devise a criterion to remove some edges of the dependency graph.

6.3.1 Motivating example

We consider a corrected version of the well-known Needham Schroeder key establishment protocol [58]. It can be described informally as follows:

1. $A \to S :\ A, B, N_a$
2. $S \to A :\ \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
3. $A \to B :\ \{K_{ab}, A\}_{K_{bs}}$
4. $B \to A :\ \{\mathsf{req}, N_b\}_{K_{ab}}$
5. $A \to B :\ \{\mathsf{rep}, N_b\}_{K_{ab}}$

We propose a modelling of this protocol in our formalism. Below, k_{as}, k_{bs}, k_{ab}, n_a, n_b, and k are names, whereas a, b, req, rep, m_1 and m_2 are constants from Σ_0.

$$P_{\mathsf{NS}} = \begin{array}{l} !\,\mathsf{new}\ c_1.\mathsf{out}(c_A, c_1).P_A \ \mid !\,\mathsf{new}\ c_2.\mathsf{out}(c_B, c_2).P_B \\ \mid !\,\mathsf{new}\ c_3.\mathsf{out}(c_S, c_3).P_S \end{array}$$

where the processes P_A, P_B, and P_S are given below.

$$\begin{array}{rl} P_A = & \mathsf{new}\ n_a. \\ & \alpha_1 : \mathsf{out}(c_1, \langle \mathsf{a}, \mathsf{b}, n_a \rangle). \\ & \alpha_2 : \mathsf{in}(c_1, \mathsf{enc}(\langle n_a, \mathsf{b}, x_{AB}, x_B \rangle, k_{as})). \\ & \alpha_3 : \mathsf{out}(c_1, x_B). \\ & \alpha_4 : \mathsf{in}(c_1, \mathsf{enc}(\langle \mathsf{req}, x_{NB} \rangle, x_{AB})). \\ & \alpha_5 : \mathsf{out}(c_1, \mathsf{enc}(\langle \mathsf{rep}, x_{NB} \rangle, x_{AB})) \\ P_B = & \beta_1 : \mathsf{in}(c_2, \mathsf{enc}(\langle y_{AB}, \mathsf{a} \rangle, k_{bs})).\mathsf{new}\ n_b. \\ & \beta_2 : \mathsf{out}(c_2, \mathsf{enc}(\langle \mathsf{req}, n_b \rangle, y_{AB})). \\ & \beta_3 : \mathsf{in}(c_2, \mathsf{enc}(\langle \mathsf{rep}, n_b \rangle, y_{AB})) \\ P_S = & \gamma_1 : \mathsf{in}(c_3, \langle \mathsf{a}, \mathsf{b}, z_{NA} \rangle).\mathsf{new}\ k_{ab}. \\ & \gamma_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle z_{NA}, \mathsf{b}, k_{ab}, \mathsf{enc}(\langle k_{ab}, \mathsf{a} \rangle, k_{bs}) \rangle, k_{as})) \end{array}$$

As in Example 2.4.3, we model the security of the exchanged key by requiring that $P^1_{\mathsf{NS}} \approx P^2_{\mathsf{NS}}$ whereP^1_{NS} and P^2_{NS} are defined as follows:

- P^1_{NS} is as the protocol P_{NS} but we add the instruction "$\beta_4 : \mathsf{out}(c_2, \mathsf{enc}(\mathsf{m}_1, y_{AB}))$" at the end of P_B;
- P^2_{NS} is as the protocol P_{NS} but we add the instruction "$\mathsf{new}\ k.\beta_4 : \mathsf{out}(c_2, \mathsf{enc}(\mathsf{m}_2, k))$" at the end of P_B.

As for the Denning Sacco protocol (see Example 2.5.3), type-compliance is satisfied. We only have to introduce some new atomic types: τ_{na}, τ_{nb}, τ_{req}, τ_{rep} and we type each constant (resp. name and variable) as expected. We denote $(\mathcal{T}_{\mathsf{NS}}, \delta_{\mathsf{NS}})$ the resulting structure-preserving typing system.

The resulting dependency graph is depicted in Figure 6.4 (and the dashed arrow is part of the dependency graph). There is no arrow due to item 3 (dependencies between outputs). As in the Denning Sacco protocol, τ_{kas} and τ_{kbs} are honest types whereas τ_{a}, τ_{b}, τ_{m}, τ_{req}, and τ_{rep} are public types. Due to the fact that τ_{kab} can not be considered as an honest type, many arrows (those that are labelled with 1.2) are added to the dependency graph. This reflects executions that will build ciphertexts with such a key.

The dependency graph is cyclic. Intuitively, this is due to the fact that the subterm at position 1.2 (the nonce N_b) outputted in α_5 may be used in input at α_4. However, if the attacker is able to access this subterm at position 1.2 in α_5, it necessarily means that he already knew this subterm already. Thus, intuitively, this dependency is not necessary. We formalise this notion in the next section. This will lead to a notion of refined dependency graph which is a dependency graph in which some arrows have been removed.

6.3.2 Appropriate marking

We first devise a general criterion to remove some of the edges of the dependency graph. We proceed by *marking* some of the positions of the graph.

Definition 6.3.1. A *marked position* of a protocol P is a pair (α, p) where $\alpha : \mathsf{out}(c, u)$ is an output action occurring in P, and p is a position of the term u. A *marking* of a protocol P is a set of marked positions of P.

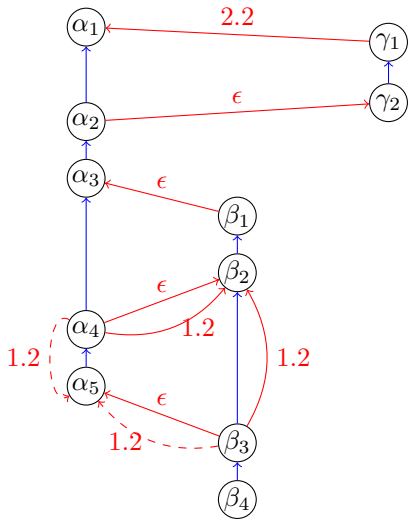


Figure 6.4: Dependency graph for P^i_{NS} w.r.t. $(\mathcal{T}_{\mathsf{NS}}, \delta_{\mathsf{NS}})$

This notion of marking is very general. We consider that a marking strategy is *appropriate* for our dependency graph if it indicates subterms that, if deducible, must be deducible earlier using a smaller set of keys.

We define a function ρ_{io} on terms very similar to the function ρ_{io} defined on types in Section 6.2.

Definition 6.3.2. Given a term t, a position p and a multiset of terms S, the function ρ_{io} is inductively defined for terms as follows:

- $\rho_{\mathsf{io}}(t_0, p, S) = \{(t_0, p)\#S\}$ for any atomic term t_0 (name, constant or variable);
- $\rho_{\mathsf{io}}(\langle t_1, t_2\rangle, p, S) = \rho_{\mathsf{io}}(t_1, p.1, S) \cup \rho_{\mathsf{io}}(t_2, p.2, S)$;
- $\rho_{\mathsf{io}}(\mathsf{enc}(t_1, t_2), p, S) =$
  $\begin{cases} \{(\mathsf{enc}(t_1, t_2), p)\#S\} \text{ if } \delta_0(t_2) \text{ is an honest type;} \\ \{(\mathsf{enc}(t_1, t_2), p)\#S\} \cup \rho_{\mathsf{io}}(t_1, p.1, S \cup \{t_2\}) \text{ otherwise.} \end{cases}$

Intuitively, $(u_0, p)\#S \in \rho_{\mathsf{io}}(u, \epsilon, \emptyset)$ if the multiset S of keys suffices to access the subterm $u_0 = u|_p$.

We show that it is always appropriate to mark a position if the corresponding subterm appears earlier in the protocol, protected by a smaller set of keys.

Definition 6.3.3. Let (α, p) be a marked position in P. (α, p) is an appropriate marked position for P if there exists an input action $\beta : \mathsf{in}(d, v)$ (or an output action $\beta : \mathsf{out}(d, v)$) in P such that:

1. $\alpha : \mathsf{out}(c, u)$ follows β in P;
2. $(u_0, p)\#S \in \rho_{\mathsf{io}}(u, \epsilon, \emptyset)$ for some u_0, and some S; and
3. $(u_0, q)\#S' \in \rho_{\mathsf{io}}(v, \epsilon, \emptyset)$ for some q, and some $S' \subseteq_{mul} S$.

A marking $\mathcal{M}$ of P is *appropriate* if all the pairs in $\mathcal{M}$ are appropriate marked positions.

Example 6.3.1. We pursue our example started in Section 6.3.1. We may set $(\alpha_5, 1.2)$ to be a marked position of P^1_{NS} (resp. P^2_{NS}). Intuitively, it is an appropriate marked position since the message $x_{NB}\sigma$ sent by the process P_A cannot be learnt at the step α_5: either $x_{NB}\sigma$ remains secret or it was learnt earlier using fewer keys. Definition 6.3.3 allows us to state that $(\alpha_5, 1.2)$ is an appropriate marked position for P^1_{NS} (resp. P^2_{NS}). Indeed, we have that:

1. $\alpha_5 : \mathsf{out}(c_1, \mathsf{enc}(\langle \mathsf{rep}, x_{NB}\rangle, x_{AB}))$ follows α_4 in P^1_{NS};
2. $(x_{NB}, 1.2)\#\{x_{AB}\} \in \rho_{\mathsf{io}}(\mathsf{enc}(\langle \mathsf{rep}, x_{NB}\rangle, x_{AB}))$; and
3. $(x_{NB}, 1.2)\#\{x_{AB}\} \in \rho_{\mathsf{io}}(\mathsf{enc}(\langle \mathsf{req}, x_{NB}\rangle, x_{AB}))$.

Definition 6.3.3 provides a simple criterion for choosing which position to mark in a protocol.

6.3.3 Refined dependency graph

We refine our dependency graph by simply removing any arrow that points towards an appropriate marked position.

Definition 6.3.4. Let P be a type-compliant protocol P (w.r.t. a structure-preserving typing system $(\mathcal{T}_P, \delta_P)$) and $\mathcal{M}$ be a marking of P. The refined dependency graph associated to P and $\mathcal{M}$ is obtained from the dependency graph of P by simply removing any arrow of the form $\alpha \rightarrow^p \beta$ for which

$$(\beta, q) \in \mathcal{M} \text{ and } q \text{ is a prefix of } p.$$

Example 6.3.2. The refined dependency graph associated to P^1_{NS} (resp. P^2_{NS}) and $\mathcal{M} = \{(\alpha_5, 1.2)\}$ is the graph depicted in Figure 6.4, when removing the dashed arrow. This dashed arrow is removed thanks to the (appropriate) marking.

Then trace equivalence is decidable for simple, type-compliant protocols, as soon as their corresponding refined dependency graph is acyclic.

Theorem 6.3.1. The problem of deciding whether two simple protocols P and Q, type-compliant w.r.t. some structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$, and with acyclic refined dependency graphs obtained relying on appropriate markings $\mathcal{M}_P$ and $\mathcal{M}_Q$ are trace equivalence (*i.e.* $P \approx Q$) is decidable.

The proof of this theorem will be the object of Section 6.5. Theorem 6.2.1 is actually a direct consequence of Theorem 6.3.1. The proof can be summarised in three steps: we first use our type-compliance assumption to focus on well-typed traces such that every message is computed as soon as possible. Then we show that every dependency is such a trace appears on the refined dependency graph of the protocols, allowing us to bound the width and depth of such a witness. Finally, we manage to bound the length of a minimal witness of non-equivalence.

6.4 Results

We review several protocols of the literature and identify whether they fall in our decidable class. We first discuss which corruption scenario is considered.

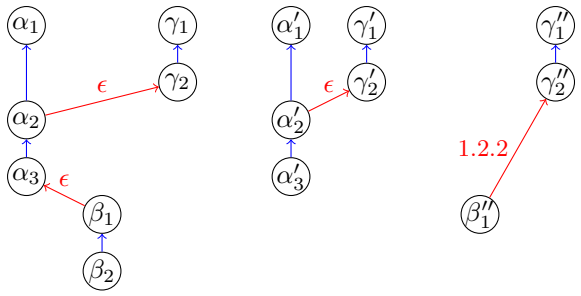


Figure 6.5: Dependency graph for P^+_{DS} w.r.t. $(\mathcal{T}^+_{\mathsf{DS}}, \delta^+_{\mathsf{DS}})$

6.4.1 Scenario with corruption

The scenario we considered so far for the Denning-Sacco protocol (as well as the Needham-Schroeder protocol) is quite simple. We only consider sessions between two honest agents a and b. Such a scenario is known to be too simplistic and some attacks may be missed, such as the well-known man-in-the-middle attack on the Needham-Schroeder public key protocol [53].

We therefore consider a scenario where honest agents are also willing to engage communications with a dishonest agent c. Let us develop this corruption scenario on the Denning-Sacco protocol. Formally, we consider P^{i+}_{DS} obtained from P^i_{DS} by adding P'_{DS} as well as P''_{DS} in parallel. The purpose of P'_{DS} is to consider that the agent a may be involved in some other sessions with a corrupted agent c, and the server S is ready to answer requests coming from them. Similarly P''_{DS} models the fact that the agent b may be involved in some sessions where the role of A is played by the corrupted agent c. Thus, we consider

$$P'_{\mathsf{DS}} = !\ \mathsf{new}\, c_1.\mathsf{out}(c'_A, c_1).P'_A \ \mid !\ \mathsf{new}\, c_3.\mathsf{out}(c'_S, c_3).P'_S$$

where P'_A and P'_S are as follows:

$$\begin{array}{rcl} P'_A & = & \alpha'_1 : \mathsf{out}(c_1, \langle \mathsf{a}, \mathsf{c}\rangle).\\ & & \alpha'_2 : \mathsf{in}(c_1, \mathsf{enc}(\langle \mathsf{b}, x'_{AB}, x'_B\rangle, k_{as})).\\ & & \alpha'_3 : \mathsf{out}(c_1, x'_B)\\ P'_S & = & \gamma'_1 : \mathsf{in}(c_3, \langle \mathsf{a}, \mathsf{c}\rangle).\ \mathsf{new}\ k'_{ab}.\\ & & \gamma'_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle \mathsf{c}, k'_{ab}, \mathsf{enc}(\langle k'_{ab}, \mathsf{a}\rangle, k_{cs})\rangle, k_{as})) \end{array}$$

We consider also:

$$P''_{\mathsf{DS}} = !\ \mathsf{new}\, c_2.\mathsf{out}(c''_B, c_2).P''_B \ \mid !\ \mathsf{new}\, c_3.\mathsf{out}(c''_S, c_3).P''_S$$

where P''_B and P''_S are as follows:

$$\begin{array}{rcl} P''_B & = & \beta''_1 : \mathsf{in}(c_2, \mathsf{enc}(\langle y''_{AB}, \mathsf{c}\rangle, k_{bs}))\\ P''_S & = & \gamma''_1 : \mathsf{in}(c_3, \langle \mathsf{c}, \mathsf{b}\rangle).\ \mathsf{new}\ k''_{ab}.\\ & & \gamma''_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle \mathsf{b}, k''_{ab}, \mathsf{enc}(\langle k''_{ab}, \mathsf{c}\rangle, k_{bs})\rangle, k_{cs})) \end{array}$$

The resulting protocols P^{1+}_{DS} and P^{2+}_{DS} are simple protocols. They are also type-compliant w.r.t. $(\mathcal{T}^+_{\mathsf{DS}}, \delta^+_{\mathsf{DS}})$ where $\mathcal{T}^+_{\mathsf{DS}}$ is an enriched version of $\mathcal{T}_{\mathsf{DS}}$ with new atomic types: τ_{c}, τ_{kcs}, $\tau_{kab'}$, and $\tau_{kab''}$. In particular, we have type-compliance for a notion of type that gives different types to k_{ab}, k'_{ab}, and k''_{ab}. The type τ_{c} is public.

The resulting dependency graph remains acyclic and is depicted in Figure 6.5. Note that there is an arrow from β_1'' to γ_2'' for the following reason. We have that

$$(\mathsf{enc}(\langle \tau_{kab''}, \tau_{\mathsf{c}} \rangle, \tau_{kbs}), 1.2.2) \in \rho_{\mathsf{out}}(u_2'' \delta^{+}_{\mathsf{DS}})$$

where u_2'' is the term occurring in the action labelled γ_2''. Intuitively, this is because the output labelled γ_2'' is an encryption with a compromised key k_{cs}, and thus the attacker could analyse this term and learn a term of type $\mathsf{enc}(\langle \tau_{kab''}, \tau_{\mathsf{c}} \rangle, \tau_{kbs})$. A term of such a type could be used to fill the input β_1''. This possible dependency is represented by an arrow from β_1'' to γ_2''. The label indicates the position at which such a term is available in the output.

The resulting dependency graph is composed of three components that are completely disconnected. This reflects the fact that the protocol ensures that there is no interaction between a session involving honest participants, and sessions that may involve some dishonest participants. Also there is no confusion between messages used at the different stages of the protocol.

For a complete corruption scenario, we then need to consider the cases where the role A is played by the agent b and the role B is played by the agent a. The resulting dependency graph is obtained by symmetry from the one displayed in Figure 6.5. It remains acyclic and is composed of six disjoint components.

In the remaining of the section, we study several symmetric key protocols of the literature and discuss whether they fall in our decidable class. For all of them, we consider the complete corruption scenario as described above on the Denning-Sacco protocol.

6.4.2 Review of symmetric key protocols

Most of the protocols we considered from [30] actually fall in our decidable class. We sometimes need them to include some explicit tags. For some of them, we need to consider a refined version of our typing graph, and we consider the one obtained using Definition 6.3.3 to mark position appropriately. Our findings are summarised in Figure 6.6. We discuss below each protocol individually.

	Dependency graph		In our
	Normal	Refined	class
Denning-Sacco	✓	✓	yes
Needham-Schroeder		✓	yes
Otway-Rees		✓	yes
Yahalom (Paulson)		✓	yes
Wide-Mouthed-Frog	✓	✓	yes
Yahalom			no
Kao-Chow (modified)		✓	yes

Figure 6.6: A ✓ means that the corresponding dependency graph is acyclic.

Denning-Sacco This protocol forms our running example and its dependency graph is acyclic, without any tagging (see Example 6.2.5).

Needham-Schroeder As discussed in Section 6.3.1, its dependency graph is not acyclic but its refined dependency graph is, even when considering the complete corruption scenario. We do not need to add explicit tags. However, contrary to what happens in the Denning-Sacco protocol, the resulting dependency graph is more complex. It is composed of six components connected with several arrows. This is due to the fact that to get type-compliance we have to give the same type to all the names that play the role of the key K_{ab} (resp. N_b) with no distinction between those involved in an honest or a dishonest session.

This can be done as follows: agents a and b are both willing to start sessions with a malicious agent c. Below, k_{as}, k_{bs}, k_{cs}, k_{ab}, k'_{ab}, k''_{ab}, n_a, n'_a, n_b, n''_b, and k are names, whereas a, b, c, req, rep, m_1 and m_2 are constants from Σ_0. As in Example 2.4.3, we model the security of the exchanged key by requiring that $P^{1+}_{\mathsf{NS}} \approx P^{2+}_{\mathsf{NS}}$ whereP^{1+}_{NS} and P^{2+}_{NS} are defined as follows:

- P^{1+}_{NS} is as the protocol P^{+}_{NS} (see below) but we add the instruction "$\beta_4 : \mathsf{out}(c_2, \mathsf{enc}(\mathsf{m}_1, y_{AB}))$" at the end of P_B;
- P^{2+}_{NS} is as the protocol P^{+}_{NS} but we add the instruction "$\mathsf{new}\, k.\beta_4 : \mathsf{out}(c_2, \mathsf{enc}(\mathsf{m}_2, k))$" at the end of P_B.

$$\begin{aligned} P^{+}_{\mathsf{NS}} = \;& !\,\mathsf{new}\; c_1.\mathsf{out}(c_A, c_1).P_A \;\mid !\,\mathsf{new}\; c_2.\mathsf{out}(c_B, c_2).P_B \\ \mid\; & !\,\mathsf{new}\; c_3.\mathsf{out}(c_S, c_3).P_S \\ \mid\; & !\,\mathsf{new}\; c_1.\mathsf{out}(c'_A, c_1).P'_A \;\mid !\,\mathsf{new}\; c_3.\mathsf{out}(c'_S, c_3).P'_S \\ \mid\; & !\,\mathsf{new}\; c_2.\mathsf{out}(c''_B, c_2).P''_B \;\mid !\,\mathsf{new}\; c_3.\mathsf{out}(c''_S, c_3).P''_S \end{aligned}$$

where processes P_A, P'_A, P_B, P''_B, P_S, P'_S, P''_S are as follows.

$$\begin{aligned} P_A = {} & \mathsf{new}\; n_a. \\ & \alpha_1 : \mathsf{out}(c_1, \langle \mathsf{a}, \mathsf{b}, n_a \rangle). \\ & \alpha_2 : \mathsf{in}(c_1, \mathsf{enc}(\langle n_a, \mathsf{b}, x_{AB}, x_B \rangle, k_{as})). \\ & \alpha_3 : \mathsf{out}(c_1, x_B). \\ & \alpha_4 : \mathsf{in}(c_1, \mathsf{enc}(\langle \mathsf{req}, x_{NB} \rangle, x_{AB})). \\ & \alpha_5 : \mathsf{out}(c_1, \mathsf{enc}(\langle \mathsf{rep}, x_{NB} \rangle, x_{AB})) \end{aligned}$$

$$\begin{aligned} P'_A = {} & \mathsf{new}\; n'_a. \\ & \alpha'_1 : \mathsf{out}(c_1, \langle \mathsf{a}, \mathsf{c}, n'_a \rangle). \\ & \alpha'_2 : \mathsf{in}(c_1, \mathsf{enc}(\langle n'_a, \mathsf{c}, x'_{AB}, x'_B \rangle, k_{as})). \\ & \alpha'_3 : \mathsf{out}(c_1, x'_B). \\ & \alpha'_4 : \mathsf{in}(c_1, \mathsf{enc}(\langle \mathsf{req}, x'_{NB} \rangle, x'_{AB})). \\ & \alpha'_5 : \mathsf{out}(c_1, \mathsf{enc}(\langle \mathsf{rep}, x'_{NB} \rangle, x'_{AB})) \end{aligned}$$

$$\begin{aligned} P_B = {} & \beta_1 : \mathsf{in}(c_2, \mathsf{enc}(\langle y_{AB}, \mathsf{a} \rangle, k_{bs})).\mathsf{new}\; n_b. \\ & \beta_2 : \mathsf{out}(c_2, \mathsf{enc}(\langle \mathsf{req}, n_b \rangle, y_{AB})). \\ & \beta_3 : \mathsf{in}(c_2, \mathsf{enc}(\langle \mathsf{rep}, n_b \rangle, y_{AB})) \end{aligned}$$

$$\begin{aligned} P''_B = {} & \beta''_1 : \mathsf{in}(c_2, \mathsf{enc}(\langle y''_{AB}, \mathsf{c} \rangle, k_{bs})).\mathsf{new}\; n''_b. \\ & \beta''_2 : \mathsf{out}(c_2, \mathsf{enc}(\langle \mathsf{req}, n''_b \rangle, y''_{AB})). \\ & \beta''_3 : \mathsf{in}(c_2, \mathsf{enc}(\langle \mathsf{rep}, n''_b \rangle, y''_{AB})) \end{aligned}$$

$$\begin{aligned} P_S = {} & \gamma_1 : \mathsf{in}(c_3, \langle \mathsf{a}, \mathsf{b}, z_{NA} \rangle).\mathsf{new}\; k_{ab}. \\ & \gamma_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle z_{NA}, \mathsf{b}, k_{ab}, \mathsf{enc}(\langle k_{ab}, \mathsf{a} \rangle, k_{bs}) \rangle, k_{as})) \end{aligned}$$

$$\begin{aligned} P'_S = {} & \gamma'_1 : \mathsf{in}(c_3, \langle \mathsf{a}, \mathsf{c}, z'_{NA} \rangle).\mathsf{new}\; k'_{ab}. \\ & \gamma'_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle z'_{NA}, \mathsf{c}, k'_{ab}, \mathsf{enc}(\langle k'_{ab}, \mathsf{a} \rangle, k_{cs}) \rangle, k_{as})) \end{aligned}$$

$$\begin{aligned} P''_S = {} & \gamma''_1 : \mathsf{in}(c_3, \langle \mathsf{c}, \mathsf{b}, z''_{NA} \rangle).\mathsf{new}\; k''_{ab}. \\ & \gamma''_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle z''_{NA}, \mathsf{b}, k''_{ab}, \mathsf{enc}(\langle k''_{ab}, \mathsf{c} \rangle, k_{bs}) \rangle, k_{cs})) \end{aligned}$$

To ensure the protocol is type-compliant, we need to consider a typing system $(\mathcal{T}^{+}_{\mathsf{NS}}, \delta^{+}_{\mathsf{NS}})$ such that x_{NB}, x'_{NB}, n_b and n''_b share the same type, τ_{nb}; x_{AB}, x'_{AB}, y_{AB}, y''_{AB}, k_{ab}, k'_{ab}, k''_{ab}, k have all type τ_{kab}; and both n_a and z_{NA} (resp. n'_a and z'_{NA}) have type τ_{na}. Which implies that $\delta^{+}_{\mathsf{NS}}(x_B) = \mathsf{enc}(\langle \tau_{kab}, \tau_{\mathsf{a}} \rangle, \tau_{kbs})$, $\delta^{+}_{\mathsf{NS}}(x'_B) = \mathsf{enc}(\langle \tau_{kab}, \tau_{\mathsf{a}} \rangle, \tau_{kcs})$ and both m_1 and m_2 must have type τ_{m}. Moreover, τ_{kas} and τ_{kbs} are honest types whereas τ_{a}, τ_{b}, τ_{c}, τ_{m}, τ_{req}, and τ_{rep} are public types.

The dependency graph for P^{i+}_{NS} (for any $i \in \{1, 2\}$) w.r.t. this scenario is then given in Figure 6.7, split into three graphs displaying edges of type 1, 2 and 3 respectively. Applying Definition 6.3.3 enables us to consider $(\alpha_5, 1.2)$ and $(\alpha'_5, 1.2)$ as appropriate marked position in P^{1+}_{NS} and P^{2+}_{NS}, which allows us to discard type 2 arrows from α_4, β_3, α'_4 and β''_3 towards α_5 and α'_5 with label 1.2.

Edges towards α'_3 (resp. γ''_2) are all with label 1.1 (resp. 1.2.2.1) and exist because both labels correspond to outputs of keys known to the agent c (encrypted by k_{cs}). As all keys generated by the server share the same type τ_{kab}, any use of such a key create an edge towards those two nodes. Similarly, as nonces created by b and outputted at labels β_2 and β''_2 share the same type τ_{nb} and are encrypted by keys of (non-honest) type τ_{kab}, inputs using such nonces all point towards these nodes. Finally, arrows with label ϵ correspond to regular executions of the protocol, albeit with some collision between messages because of the previously detailed equal types.

Otway-Rees The tagged version of the Otway-Rees protocol can be informally described as follows.

$$
\begin{array}{l}
A \to B : M, A, B, \{1, N_a, M, A, B\}_{K_{as}} \\
B \to S : M, A, B, \{1, N_a, M, A, B\}_{K_{as}}, \{2, N_b, M, A, B\}_{K_{bs}} \\
S \to B : M, \{3, N_a, K_{ab}\}_{K_{as}}, \{4, N_b, K_{ab}\}_{K_{bs}} \\
B \to A : M, \{3, N_a, K_{ab}\}_{K_{as}}
\end{array}
$$

Note that, considering a scenario with no corruption, its untagged version can be shown to be simple and type-compliant by typing k_{ab} with the same type as $\langle m, \mathsf{a}, \mathsf{b}\rangle$. However, its dependency graph would be cyclic (a cycle will appear between the two actions of the role S). We therefore consider the tagged version of the Otway-Rees protocol. Its dependency graph is still cyclic but becomes acyclic when marking several positions. In particular, we have to mark

1. all the positions (in roles B and S) where M appears in plaintext position;
2. the positions (in roles A and B) at which the variables modelling ciphertext forwarding occur; and
3. the positions in roles S (only those that involve the dishonest agent c) that correspond to N_a (in case c plays the role A) and N_b (in case c plays the role B).

The fact that this marking strategy is appropriate is a direct consequence of Definition 6.3.3.

Similarly, the *Wide-Mouthed-Frog* and the *Yahalom (Paulson version)* protocols need to be explicitly tagged. Their refined dependency graphs are acyclic. Actually the normal dependency graph of the Wide-Mouthed-Frog protocol is already acyclic.

Yahalom Consider now the original Yahalom protocol. Its tagged version can be informally described as follows.

$$
\begin{array}{ll}
1. & A \to B : \ A, N_a \\
2. & B \to S : \ B, \{1, A, N_a, N_b\}_{K_{bs}} \\
3. & S \to A : \ \{2, B, K_{ab}, N_a, N_b\}_{K_{as}}, \{3, A, K_{ab}\}_{K_{bs}} \\
4. & A \to B : \ \{3, A, K_{ab}\}_{K_{bs}}, \{4, N_b\}_{K_{ab}}
\end{array}
$$

When considering only honest sessions, the corresponding dependency graph is acyclic. However, cycles appear if sessions with dishonest agents are considered. Intuitively, this is due to the fact that the nonce N_b sent at the final step is encrypted under the key K_{ab} which secrecy cannot be statically guaranteed. Therefore, the nonce N_b could potentially be learnt at step 4 and be reused earlier (at Step 2 for example). Note that such a protocol would be declared leaky in [47].

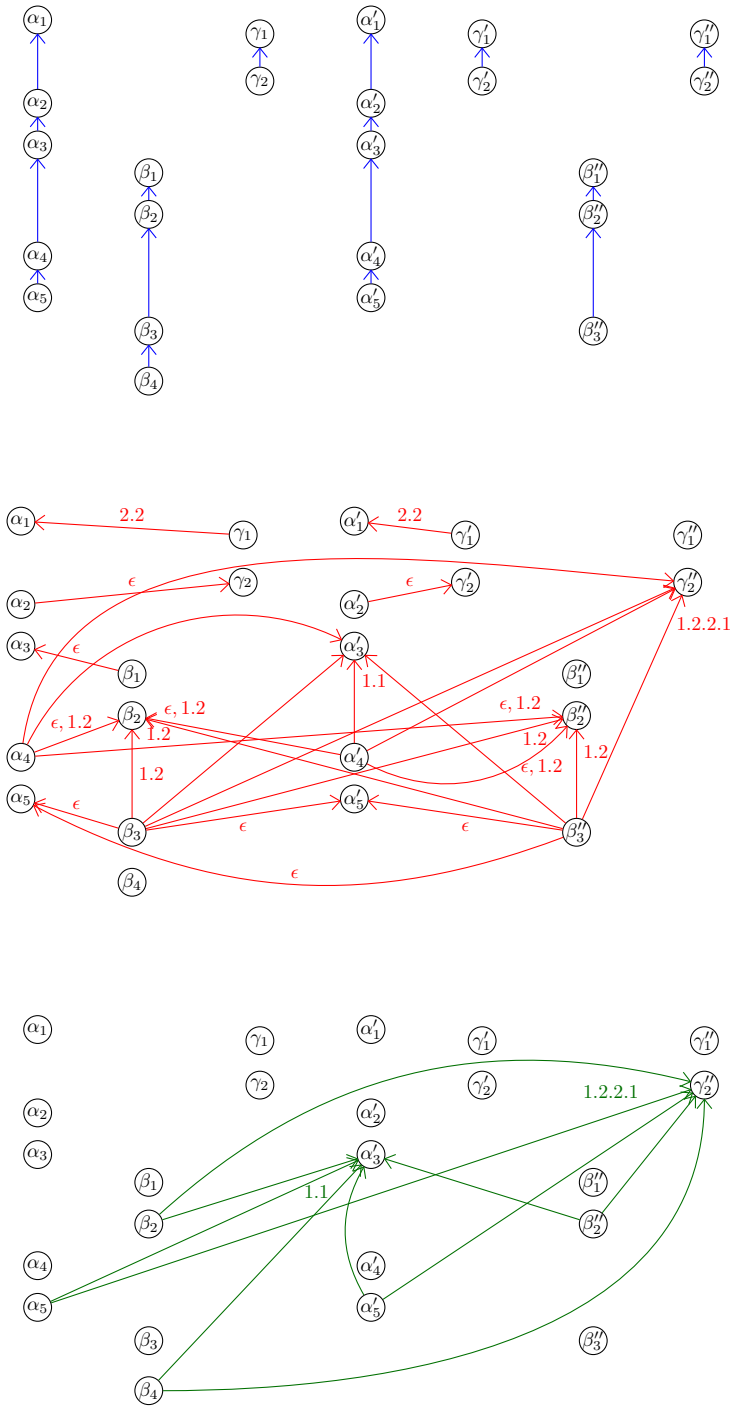


Figure 6.7: Dependency graph for P^{i+}_{NS} w.r.t. $(\mathcal{T}^{+}_{\mathsf{NS}}, \delta^{+}_{\mathsf{NS}})$

Kao-Chow Again, it is necessary to tag this protocol to obtain type-compliance w.r.t. a relatively fine-grained typing system, and to avoid some cycles in the dependency graph.

$$
\begin{array}{l}
1.\ A \to S :\ A, B, N_a \\
2.\ S \to B :\ \{1, A, B, N_a, K_{ab}\}_{K_{as}}, \{2, A, B, N_a, K_{ab}\}_{K_{bs}} \\
3.\ B \to A :\ \{1, A, B, N_a, K_{ab}\}_{K_{as}}, \{3, N_a\}_{K_{ab}}, N_b \\
4.\ A \to B :\ \{4, N_b\}_{K_{ab}}
\end{array}
$$

Even with explicit tags, the resulting dependency graph contains a cycle due to the fact that, at the third step, B sends the nonce N_a under K_{ab}, which security cannot be statically asserted. However, N_a is intuitively public. We therefore slightly modify this protocol, assuming that S also sends N_a in clear at the second step, which suffices to obtained an acyclic (refined) dependency graph using Definition 6.3.3. This is a typical example where another marking strategy could be applied to (soundly) obtain an acyclic graph.

6.4.3 Detailed comparison with [47]

Sibylle Fröschle has recently obtained [47] a decidability result for an unbounded number of sessions, for the property of "leakiness". Our decidability result differs from Fröschle's result on several general points, in particular in terms of primitives and properties that can be handled. Strictly speaking the two results are incomparable since we study trace equivalence while [47] analyses "leakiness", a specially designed property that implies secrecy. In this section, we highlight the main similarities and differences of the two approaches.

Primitives. Fröschle's result applies to all standard cryptographic primitives (concatenation, symmetric and asymmetric encryption, hash, and signatures) while we only consider concatenation and symmetric encryption. This is due to the fact that our decidability result builds upon [24] (to limit ourselves to well-typed traces), which scope is limited to concatenation and symmetric encryption.

Properties. We consider more general security properties since we can decide any equivalence-based property (provided the processes fall into our class), while [47] only applies to the particular leakiness property. Leakiness enforces that a data (a nonce or a key) is either immediately deducible or secret. Note that leakiness is strictly stronger than secrecy. It disallows protocols with temporary secrets but it also discards some very reasonable protocols such as the Needham-Schroeder symmetric key protocol, due to parallel sessions between honest and dishonest agents. Indeed, assume A initiates (honestly) a session with C. Then the key K_{ac} generated by the server is not immediately deducible to the attacker since it is protected by K_{as} but will be deducible as soon as A forwards it to C under the key K_{cs}. The protocol will be declared leaky although no one cares about the secrecy of K_{ac}.

Dependency graph. One important common point between [47] and our result is the notion of dependency graph that should be acyclic. The graph defined in [47] reflects sequential dependencies similarly to our dependency graph. Regarding data dependencies, there is an edge between two actions as soon as their corresponding terms can be instantiated such that they share a common ciphertext as subterm. As a consequence, acyclicity can only be satisfied in a typed model. Therefore, [47] assumes that agents can recognise the type of a data, *e.g.* do not confuse a nonce with a ciphertext or a pair of nonces. Fröschle's result cannot consider protocols with ciphertext forwarding. In some cases where ciphertexts are just appended to the rest of the message, [47] devises a simple transformation. However, this transformation does not apply to protocols including more involved ciphertext forwarding such as the Denning-Sacco and the Needham-Schroeder protocols.

The graph considered in [47] is somewhat simpler (*i.e.* contains less arrows). In particular, this graph does not consider key dependencies (item 3 of Definition 6.2.2). This is due to the leakiness property: there is no temporary secret thus a key is either secret or public.

6.5 Proof of our decidability results

We prove our decidability results (Theorems 6.2.1 and 6.3.1) in three main steps. In both cases, we bound the length of a witness of non-equivalence, and then conclude by invoking a decidability result for a bounded number of sessions (*e.g.* [24]).

Given two simple protocols P and Q, a *witness of non-inclusion* for $P \not\sqsubseteq Q$ is a trace tr for which there exists ϕ such that $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and:

- either there does not exist ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$,
- or such a ψ exists and $\phi \not\sim \psi$.

A *witness of non-equivalence* is a trace tr that is a witness for $P \not\sqsubseteq Q$ or $Q \not\sqsubseteq P$.

Note that for a simple protocol, once the sequence tr is fixed, all the frames reachable through tr are actually in static equivalence, which ensures the unicity of ψ, if it exists, up-to static equivalence.

The three main steps of our proof can be summarised as follows:

1. We first rely on our type-compliance assumption. We show that we can restrict our attention to witnesses that are well-typed and we further show that each message occurring in such a trace can be computed *as soon as possible* (asap) (see Lemmas 6.5.1 and 6.5.2). Intuitively, recipes should refer to messages that occur as early as possible.
2. Then, we show that all the dependencies occurring in such a well-typed and asap trace comply with the dependency graph. Hence, we bound the *width* (Lemma 6.5.7) as well as the *depth* (Corollary 6.5.1) of such a witness exploiting the acyclicity of our dependency graph.
3. Lastly, we explain how to bound the length of a minimal witness (Lemma 6.5.9).

6.5.1 Reducing equivalence

We first show that we may focus on pseudo-well-typed witnesses of attacks. Formally, we cast Theorem 4.1.1 in our framework.

Theorem 6.5.1 (Theorem 4.1.1 revisited)**.** Let P and Q be two simple protocols type-compliant w.r.t. some structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$. We have that $P \not\approx Q$ if, and only if, there exists a witness of non-equivalence tr such that:

- either $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ for some ϕ and (tr, ϕ) is pseudo-well-typed w.r.t. $(\mathcal{T}_P, \delta_P)$.
- or $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ for some ψ and (tr, ψ) is pseudo-well-typed w.r.t. $(\mathcal{T}_Q, \delta_Q)$.

Moreover, there is no $\mathsf{c} \in \Sigma_0$ having an honest type occurring in key position in tr.

Proof. Compared to Theorem 4.1.1, we consider structure-preserving typing system, instead of typing systems, and we moreover need to make sure that there exists no constant with an honest type occurring in key position in a pseudo-well-typed witness of non-equivalence. Lemma 2.5.1 ensures that $(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$ are typing systems. Then Theorem 4.1.1 ensures that if there exists a witness of non-equivalence tr of $P \not\approx Q$, the first two items of Definition 4.1.1 hold. We still need to show that for any $\mathsf{c} \in \Sigma_0$ such that $\delta(\mathsf{c})$ is honest, c does not appear in key position in $\mathsf{tr}\phi\!\downarrow$. From Lemma 4.1.1, $\mathsf{tr}\phi\!\downarrow = \mathsf{tr}_s\sigma$ for some first-order trace tr_s and substitution σ well-typed (by Definition 4.1.1). The proof of Theorem 4.2.1 ensures $ESt(\mathsf{tr}\phi\!\downarrow) \subseteq ESt(\mathsf{tr}_s)\sigma$ where ESt denotes the set of encrypted subterms of its argument (defined for terms and naturally extended to traces). Let $\mathsf{enc}(u, v) \in ESt(\mathsf{tr}\phi\!\downarrow)$. There exists $\mathsf{enc}(u', v') \in ESt(\mathsf{tr}_s)$ such that $\mathsf{enc}(u, v) = \mathsf{enc}(u', v')\sigma = \mathsf{enc}(u'\sigma, v'\sigma)$. Because keys are atomic

in our model, v and v' are atoms. Suppose *ad absurdum* $v \in \Sigma_0$ and $\delta_P(v)$ is honest. Because σ is well-typed, $\delta_P(v') = \delta_P(v)$. If v' were a variable or a constant in Σ_0, $\delta_P(v')$ cannot be honest, it would be in contradiction with item (ii) of the definition of honest type. Then v' has to be a name, causing $v = v'\sigma = v'$ to be a name too; contradiction once again. Thus no term in a key position in $\mathsf{tr}\phi\downarrow$ can be an element of Σ_0 and of an honest type. Hence (tr, ϕ) is indeed pseudo-well-typed. □

To bound the length of a witness, it is important to avoid some unnecessary detours, and this is the purpose of computing messages as soon as possible.

Definition 6.5.1. Given a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and a message m such that $\phi \vdash m$. We say that R is an *asap recipe* of m if:

- $R\phi\downarrow = m$,
- $R = C[R_1, \ldots, R_n]$ where C contains only constructors, for any i, R_i contains only destructors and $R_i\phi\downarrow$ is not a pair,
- R is the minimal recipe satisfying those conditions, w.r.t. to the following order: for any $\mathsf{w}, \mathsf{w}' \in dom(\phi)$, $\mathsf{w} < \mathsf{w}'$ if w occurs before w' in ϕ, and for any recipe R, R', $R < R'$ if $vars^{\#}(R) <_{mul} vars^{\#}(R')$; where $<_{mul}$ is the multiset extension of $<$ and $vars^{\#}(R)$ denotes the multiset of variables in R.

We say that a trace (tr, ϕ) of a protocol P is an *asap trace* if for any input recipe R occurring in tr, we have that R is an asap recipe of $R\phi\downarrow$ w.r.t. (tr, ϕ).

Example 6.5.1. The trace (tr, ϕ) given in Example 2.3.1 is not an asap trace. Indeed, $\mathsf{in}(ch_3, \mathsf{w}_1)$ occurs in tr and $\mathsf{w}_1\phi\downarrow = \langle \mathsf{a}, \mathsf{b} \rangle$. Thus w_1 is a recipe of $\langle \mathsf{a}, \mathsf{b} \rangle$. However, it is not an asap recipe since $\langle \mathsf{a}, \mathsf{b} \rangle$ is deducible from the empty frame (remember that a and b are public constants from Σ_0). For the same reason, (tr', ϕ_1') and (tr', ϕ_2') are not asap traces.

In the following lemma, we prove it is always possible to find an asap recipe which reduces to a deducible term in any given frame.

Lemma 6.5.1. Given a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and a message t such that $\phi \vdash t$, there exists an asap recipe R of t.

Proof. As $\phi \vdash t$, there exists a recipe R such that $R\phi\downarrow = t$. Let us consider the rewriting rules

$$\mathsf{dec}(\mathsf{enc}(x, y), z) \Rightarrow x$$
$$\mathsf{proj}_i(\langle x_1, x_2 \rangle) \Rightarrow x_i$$

and for any recipe R, let $R\Downarrow$ be the normal form for the $\Rightarrow$ rewriting. As t is a message, from Lemma A.1.2, we get that $R\Downarrow\phi\downarrow = t$. $R\Downarrow$ is of the form $C[R_1, \ldots, R_n]$ where C contains only constructors and for any i, R_i only destructors (without being asap at this point) as keys are atomic in our model. At this point, we proved that given a frame ϕ and t such that $\phi \vdash t$, there exists a recipe R of t such that $R = C[R_1, \ldots, R_n]$ where C contains only constructors; for any i, R_i contains only destructors.

Furthermore, to ensure $R_i\phi\downarrow$ is not a pair, for any recipe $M = C[R_1, \ldots, R_n]$ such that C is constructor-only, R_i destructor-only, consider the following induction on $\sum_{i=1}^{n} |R_i\phi\downarrow|$; where $|t|$ denotes the size of the term t:

- if $R_i\phi\downarrow$ is an atom or encryption for any i: we have the final result.
- if $R_i\phi\downarrow = \langle s, t \rangle$ for some i: applying the already proven result on $\mathsf{proj}_1(R_i)$ (resp. $\mathsf{proj}_2(R_i)$) gives a recipe $R_i' = C_i'[R_1', \ldots, R_p']$ (resp. $R_i'' = C_i''[R_1'', \ldots, R_q'']$) such that C_i' (resp. C_i'') is constructor-only and each R_j' (resp. R_j'') contains only destructors. $M' = C[R_1, \ldots, \langle R_i', R_i'' \rangle, \ldots, R_n]$ is made of constructor on top of destructors and $\sum_{j \neq i} |R_j\phi\downarrow| + |R_i'\phi\downarrow| + |R_i''\phi\downarrow| < \sum_{i=1}^{n} |R_i\phi\downarrow|$.

An asap recipe is hence obtained by considering a minimal element among the recipes satisfying those properties. □

Trace equivalence can be equivalently redefined by considering only asap recipes in Definition 2.4.2. We expose here a definition for *well-typed asap trace equivalence* and prove in Lemma 6.5.2, its equivalence with traditional trace equivalence for the protocols we consider. The intuition behind asap traces is to consider only input messages that are built as soon as they can be deducible. In particular, with asap recipes, the attacker cannot take detours to deduce a particular message. Yet it does not hinder the power of the attacker as static equivalence still ensures that recipes leading to the same term in one frame must do the same in the other frame.

Definition 6.5.2 (well-typed asap trace equivalence)**.** Two protocols P and Q type-compliant w.r.t. their respective structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$ are in *well-typed asap trace equivalence*, denoted $P \approx' Q$ if, and only if:

- for every asap trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ which is pseudo-well-typed w.r.t. (P, δ_P) there exists a trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ such that $\phi \sim \psi$,
- for every asap trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ which is pseudo-well-typed w.r.t. (Q, δ_Q) there exists a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ such that $\psi \sim \phi$.

We now prove that this variant of trace equivalence matches, for our class of protocols, to the original definition of trace equivalence. To do so, we use Theorem 6.5.1 to focus only on pseudo-well-typed trace and then show that considering only asap trace does not affect the ability for the attacker to distinguish between two protocols.

Lemma 6.5.2. Let P and Q be two protocols type-compliant w.r.t. their respective structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$. Then $P \approx Q$ if, and only if, $P \approx' Q$.

Proof. The direct implication is direct. Let us then assume that $P \not\approx Q$ and prove that $P \not\approx' Q$. Theorem 6.5.1 enables us to only consider a pseudo-well-typed witness of non-equivalence. The proof is then carried by induction on the length of the minimal witness of non-equivalence. Note that making a trace asap does not change its first-order values. □

We are then able to show that, when looking for an attack, *i.e.* a witness of non-equivalence, we can further restrict our attention to consider pseudo-well-typed witnesses that are also asap.

6.5.2 Exploiting the dependency graph

Given an asap and pseudo-well-typed execution trace (tr, ϕ) of a simple protocol P, we can see it as a dag D (directed acyclic graph) whose vertices are actions of tr, and edges represent sequential dependencies and data dependencies. Note that such a dag can be computed simply from tr since sequential dependencies may be inferred from the channel names occurring in tr, and data dependencies are inferred from input recipes that occur in tr.

Our ultimate goal is to bound the length of tr, and thus the number of vertices in D. We first show that we are able to bound its depth and its width.

The first lemma links the actions of any trace of the protocol with the actions of the protocol themselves.

Lemma 6.5.3. Given a simple protocol P, $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, and an action a appearing in tr, there exists a unique action in P which corresponds to the execution of a in tr.

Proof. Direct consequence of our semantics and the use of fresh channel names which identify actions in the trace. □

We can now define an execution graph and ensure that, for simple protocols, any trace corresponds to a unique execution graph.

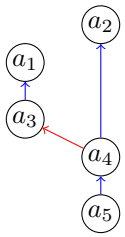


Figure 6.8: Execution graph D of tr w.r.t. P_{DS}

Definition 6.5.3. Let P be a protocol, $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$. An *execution graph* of tr w.r.t. P is a directed acyclic graph D whose vertices are the actions of tr and whose edges are defined as follows:

- there is an edge from $\mathsf{in}(c, R)$ to an action $\mathsf{out}(d, \mathsf{w})$ if $\mathsf{w} \in vars(R)$;
- there is an edge from an action a_2 to an other action a_1 if the action corresponding to a_2 in P directly follows the action corresponding to a_1 in P.

Note that such an action is uniquely defined thanks to Lemma 6.5.3.

Example 6.5.2. Consider the trace

$$\mathsf{tr} = \mathsf{out}(c_A, ch_1).\mathsf{out}(c_S, ch_3).\mathsf{out}(ch_1, \mathsf{w}_1).\mathsf{in}(ch_3, \mathsf{w}_1).\mathsf{out}(ch_3, \mathsf{w}_2)$$

and

$$\phi = \{\mathsf{w}_1 \ \triangleright \langle \mathsf{a}, \mathsf{b} \rangle, \mathsf{w}_2 \ \triangleright \mathsf{enc}(\langle \mathsf{b}, k_{ab}, \mathsf{enc}(\langle k_{ab}, \mathsf{a} \rangle, k_{bs}) \rangle, k_{as}), \}.$$

We have that $(\mathsf{tr}, \phi) \in \mathsf{trace}(P_{\mathsf{DS}})$ (see Example 6.1.1). For the sake of clarity, let us define a_i to be the i-th action of tr. Then $\mathsf{tr} = a_1.a_2.a_3.a_4.a_5$. Let D be the directed acyclic graph defined in Figure 6.8: D is an execution graph D of tr w.r.t. P_{DS}. Similarly to what we did with dependency graphs, we drew in in red edges of the first kind, corresponding to data dependencies and drew in blue edges of the second kind, corresponding the sequential dependencies.

Lemma 6.5.4. If P is a simple protocol and $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, then there exists a unique execution graph of tr w.r.t. P. Moreover, if Q is a simple protocol and there exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$, then the execution graph of tr w.r.t. Q is equal to the execution graph of tr w.r.t. P.

Proof. We prove that we can build the execution graph by looking solely at tr in the case of simple protocols: data dependencies are explicit through the recipes, while sequential dependencies can be deduced from the channel c' of an action, $\mathsf{new}\ c'.\mathsf{out}(c, c')$ the action that created it, and the position of this action in the sequence of all actions on channel c'. Hence, we have that tr completely and uniquely defines an execution graph for tr w.r.t. P. Moreover, as the dependencies depend on tr alone, the graph does not vary depending on whether tr is a trace of P or Q. □

Bounding the depth of D

Intuitively, any sequential and data dependency occurring in a well-typed and asap trace is already present in the dependency graph. This is obvious regarding sequential dependencies since all these dependencies have been added in the dependency graph. However, regarding data dependencies, this result strongly relies on the fact that we consider well-typed and asap traces.

In this first lemma we justify the appellations of *honest* and *public* by their consequences in terms of deducibility for terms of such a type.

Lemma 6.5.5. Let P be a protocol type-compliant w.r.t. a structure-preserving typing system $(\mathcal{T}, \delta)$.

- If τ is an honest type, n a name such that $\delta(n) = \tau$, then for any well-typed trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, we have that $\phi \not\vdash n$.
- If τ is a public type, t a term such that $\delta(t) = \tau$, then for any well-typed $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, $\phi \vdash t$ implies $\phi_0 \vdash t$; where ϕ_0 denotes the empty frame.

Proof. We consider the two items separately:

- As τ is honest, τ does not appear in plaintext position in $u\delta_P$ for any u in P. Suppose *ad absurdum* there exists $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ pseudo-well-typed such that $\phi \vdash n$. There exists a recipe R with only destructors (see Lemma 6.5.1; as n is atomic, the context C is empty) such that $R\phi\downarrow = n$. Let w be the leaf of R at the leftmost position (*i.e.* at the longest position $p = 11\ldots1$ such that $R|_p$ is defined): n appears in plaintext position of $\mathsf{w}\phi$. Because ϕ is well-typed, $\delta_P(\mathsf{w}\phi) = u\delta_P$ for some output u in P. But then, as n is a subterm of $\mathsf{w}\phi$ in plaintext position, $\delta_P(n)$ is a subterm of $u\delta_P$ in plaintext position: thus τ appears in plaintext position in some $u\delta_P$ with u in P. Contradiction with τ being an honest type. Hence $\phi \not\vdash n$.
- Suppose $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ is a well-typed trace such that $\phi \vdash t$. Let a be a leaf of t: a is an atom and can be an element of Σ_0 (constant) or $\mathcal{N}$ (name). Let us consider the latter: because names are initially unknown to the attacker, $\phi_0 \not\vdash a$ and $\delta_P(a) = \delta_P(n)$ for some name n in P. The name a being a subterm of t, $\delta_P(n)$ is a subterm of $\delta_P(t) = \tau$ and thus τ would not be public. Hence every leaf of t is a constant in Σ_0 and so $\phi_0 \vdash t$.

□

As we did after Definition 6.2.1, from Definition 6.3.2, we extend the definition of ρ_{in} and ρ_{out} to terms.

$$\rho_{\mathsf{out}}(t') = \{(t,p) \mid (t,p)\#S \in \rho_{\mathsf{io}}(t', \epsilon, \emptyset)\};\text{ and}$$
$$\rho_{\mathsf{in}}(t') = \{t, t_1, \ldots, t_k \mid (t,p)\#\{t_1, \ldots, t_k\} \in \rho_{\mathsf{io}}(t', \epsilon, \emptyset)\}.$$

Intuitively, $\rho_{\mathsf{out}}(t')$ returns the terms that may be deducible by the attacker once t' is outputted, whereas $\rho_{\mathsf{in}}(t')$ returns all the terms that may be used by the attacker to fill an input with term t'. Next, Lemma 6.5.6 ensures these definitions are consistent with Definition 6.2.1 when considering the type $\delta(t)$ of any term t.

Lemma 6.5.6. Let P be a protocol type-compliant w.r.t. a structure-preserving typing system $(\mathcal{T}, \delta)$. For any $u, t \in \mathcal{T}_0(\Sigma_{\mathsf{c}}, \Sigma_0 \cup \mathcal{N})$ such that no $\mathsf{c} \in \Sigma_0$ occurs in key position in t and for any position p:

- $(u,p)\#S \in \rho_{\mathsf{io}}(t, \epsilon, \emptyset) \Rightarrow (\delta(u),p)\#\delta(S) \in \rho_{\mathsf{io}}(\delta(t), \epsilon, \emptyset)$; where $\delta(S) = \{\delta(s) | s \in S\}$
- $(u,p) \in \rho_{\mathsf{out}}(t) \Rightarrow (\delta(u),p) \in \rho_{\mathsf{out}}(\delta(t))$
- $u \in \rho_{\mathsf{in}}(t) \Rightarrow \delta(u) \in \rho_{\mathsf{in}}(\delta(t))$

Proof. The typing function of a structure-preserving typing system $(\mathcal{T}, \delta)$ behaves like a substitution and therefore the two definitions of ρ_{io}, defined through inductions on terms are compatible. t (resp. u) and $\delta(t)$ (resp. $\delta(u)$) share a similar structure (any position defined in t is defined in $\delta(t)$) and any key used to open an encryption in t, and thus deducible, cannot be a constant by hypothesis nor be honest (as per Lemma 6.5.5) and thus the corresponding encryption in $\delta(t)$ is also opened. □

The next proposition is one of our key results. It requires to control data dependencies and in particular data dependencies that may occur due to keys: it may be necessary to decrypt to obtain a new key that in turn will be used to learn another key and so on. We show that our dependency graph actually captures all dependencies. When refining the dependency graph with appropriate marking, the main idea is that edges that are removed correspond to dependencies that can not happen in any asap trace.

Proposition 6.5.1. Let P be a simple protocol type-compliant w.r.t. some structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$. Let $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ be an asap and pseudo-well-typed trace w.r.t. $(\mathcal{T}_P, \delta_P)$.

For any pair of actions $\mathsf{in}(d, R)$ / $\mathsf{out}(c, \mathsf{w})$ occurring in tr with $\mathsf{w} \in vars(R)$, we have that $\alpha \rightarrow_G^+ \beta$ where:

- $\alpha, \beta \in \mathcal{L}(P)$ are the labels associated to the actions $\mathsf{in}(d, R)$ and $\mathsf{out}(c, \mathsf{w})$ respectively;
- $\rightarrow_G^+$ is the transitive closure of the relation $\rightarrow$ in the dependency graph G associated to P.

Proof. Let a/b be such a pair of actions in tr: b corresponds to an output action $\mathsf{out}(c, \mathsf{w})$ and a to an input action $\mathsf{in}(d, R)$ in tr with $\mathsf{w} \in vars(R)$.

As tr is asap, $R = C[R_1, \ldots, R_n]$ for some C which contains only pairs and encryption and such that for every $i \in \{1, \ldots, n\}$, R_i is made of projections and decryption only; and moreover $R_i\phi\!\downarrow$ is either an atom or an encrypted subterm. As such, for every $\mathsf{w} \in dom(\phi)$, either w appears at the leftmost position of R_i for some i; or w is used as a decryption key (*i.e.* w always appears as the second argument of some decryption).

Let us consider the first case for w: because $R_i\phi\!\downarrow$ is a subterm of $\mathsf{w}\phi$ and w occurs as a plaintext operand in R_i, $R_i\phi\!\downarrow \in \rho_{\mathsf{out}}(\mathsf{w}\phi)$; and similarly $R_i\phi\!\downarrow$ is a subterm of $R\phi\!\downarrow$ (as C is constructor-only), leading to $R_i\phi\!\downarrow \in \rho_{\mathsf{in}}(R\phi\!\downarrow)$ (see Lemma 6.5.5: ϕ does not contain any constant of Σ_0 in key position, deducible keys have then to be nonces, which cannot be honestly typed if deducible). As such there exist two positions p, q such that $\mathsf{w}\phi|_p = R\phi\!\downarrow|_q = t$ for some i and some term t which is either an encrypted term, a name or a constant. .

Let u (resp. v) be the term occurring at label β (resp. α) in P. There exist two ground substitutions σ_u, σ_v such that $u\sigma_u|_p = t = v\sigma_v|_q$. Because (tr, ϕ) is well-typed, $\delta_P(R\phi\!\downarrow) = \delta_P(v)$ as they share the same type, and similarly, $\delta_P(\mathsf{w}\phi) = \delta_P(u)$. Because a structure-preserving typing system is a substitution (whose domain is extended to elements of $\Sigma_0 \cup \mathcal{N}$) and thus compatible with subterms (*i.e.* $\delta_P(u|_p) = \delta_P(u)|_p$ for any term u and position p which is defined for u), we get that $\delta_P(R\phi\!\downarrow|_q) = \delta_P(v)|_q$ and $\delta(\mathsf{w}\phi|_p) = \delta_P(u)|_p$. Thus $\delta_P(u)|_p = \delta_P(v)|_q$, which can be rewritten as $u\delta_P|_p = v\delta_P|_q$ (using the traditional substitution notation). From Lemma 6.5.6, $u\delta_P|_p \in \rho_{\mathsf{out}}(u\delta_P)$ and $v\delta_P|_q \in \rho_{\mathsf{in}}(v\delta_P)$. It follows then, by Definitions 6.2.2 and 6.3.4, that an edge could exist from α to β. If it is the case, we are done. If $u\delta_P|_p$ is public type, Lemma 6.5.5 ensures $\phi_0 \vdash u\sigma_u|_p$, which contradicts the fact that R is an asap recipe (as one of its subterms is not asap). Otherwise there exists a appropriate marked position (β, p') such that p' is a prefix of p. As such, because there exists $i \in \{1, \ldots, n\}$ such that $R_i\phi\!\downarrow = u\sigma_u|_p$, there exists a subterm R_i' of R_i such that $R_i'\phi\!\downarrow = u\sigma_u|_{p'}$ (as p' is a prefix of p and R_i is made of destructors only). According to Definition 6.3.3, there exists an action $\gamma : \mathsf{in}(d, u')$ (or $\gamma : \mathsf{out}(d, u')$) such that β follows γ, $(u|_{p'}, p')\#S \in \rho_{\mathsf{io}}(u, \epsilon, \emptyset)$ for some multiset S; and $(u|_{p'}, q)\#S' \in \rho_{\mathsf{io}}(u', \epsilon, \emptyset)$ for some q, and some $S' \subseteq_{mul} S$. Note that because marked positions are defined on the specification of the protocol P, $u|_{p'}$ is well-defined and $u|_{p'}\sigma_u = u\sigma_u|_{p'}$. In particular, $\phi \vdash u\sigma_u|_{p'}$ using a destructor-only recipe R_i' with w in its leftmost position implies that for every $k \in S$, there exists a subterm R^k of R_i' such that $R^k\phi\!\downarrow = k$. As γ occurs before β in P, let R'' be the recipe used with this action $\gamma : \mathsf{in}(d, u')$ (or $\gamma : \mathsf{out}(d, u')$) in tr. As tr is asap, R'' is asap and $R''\phi\!\downarrow = u'\sigma_u$. As $(u|_{p'}, q)\#S' \in \rho_{\mathsf{io}}(u', \epsilon, \emptyset)$, there exists a destructor-only context $R_1''[\_]$ with a hole in its leftmost leaf position such that $R_1''[R'']\phi\!\downarrow = u'|_q\sigma_u = u|_{p'}\sigma_u$ and $vars^{\#}(R_1'') = \bigcup_{k \in S'} vars^{\#}(R^k) \subseteq \bigcup_{k \in S} vars^{\#}(R^k)$ as $S' \subseteq_{mul} S$ and by using the R^k as decryption keys when needed. Note that in this context $vars(R)$ denotes the multiset of variables occurring in R. As R'' is asap and $R''\phi\!\downarrow$ is a message, $R_1''[R'']\Downarrow$ is a recipe made of a context of constructors on top of destructor-only recipes too (see the definition of $\Downarrow$ in the proof of Lemma 6.5.1). Lemma A.1.2 ensures $R_1''[R'']\Downarrow\phi\!\downarrow = u|_{p'}\sigma_u$. Because γ occurred before β, $vars^{\#}(R'') <_{mul} \{\mathsf{w}\}$, which leads to $vars^{\#}(R_1''[R'']\Downarrow) < vars^{\#}(R_i')$ and implies that R_i' is not minimal. R_i' not being minimal implies R_i is not either. Then R would not be asap; and hence $\alpha \rightarrow \beta$.

Now consider the second case for w. Let w_0 be the variable at the leftmost position in R_i (*i.e.* $R_i|_{p_0} = \mathsf{w}_0$ where p_0 is the longest position 1^k such that $R_i|_{p_0}$ is defined). Note that $\mathsf{w}_0 \neq \mathsf{w}$ as otherwise we would be in the first case. For position p, we define w_p to be the leaf of R_i at position $p.1 \ldots 1$, α_P the label at which the output of w_p took place, u_p the output term at label α_p in (the specification) P and σ_p the substitution such that $\mathsf{w}_p\phi = u_p\sigma_p$. As $R_i|_p\phi\!\downarrow$ is a subterm of $\mathsf{w}_p\phi$, there exists a position $\tilde{p}$ such that $R_i|_p\phi\!\downarrow = u_p\sigma_p|_{\tilde{p}}$. We then define a number of sets of positions as

follows: for any $j \in \mathbb{N}^*$

$$K_j = \{p \text{ s.t. } R_i|_p \text{ is defined and } \exists m_1, \dots m_j \in \mathbb{N}, p = 1^{m_1}.2 \dots 1^{m_j}.2\}$$

Note that because R_i is finite, there exits $j_0 \in \mathbb{N}$ such that $\forall j > j_0, K_j = \emptyset$. In particular, for any $p \in K_1$, $R_i|_p\phi\downarrow$ corresponds to a subterm of $\mathsf{w}_p\phi$ and an atom (as keys are atomic) used to decrypt the first layers of encryption of $\mathsf{w}_0\phi$. It implies that $R_i\phi\downarrow$ appears as a plaintext (or subterm of plaintext) of encryptions by $R_i|_p\phi\downarrow$ for any $p \in K_1$. Similarly, for any $p \in K_{j+1}$, $R_i|_p\phi\downarrow$ is a subterm of $\mathsf{w}_p\phi\downarrow$ and an atomic key used to decrypt some $R_i|_q\phi\downarrow$ where $q \in K_j$.

More formally, for any j and any $p \in K_{j+1}$, there exists $q \in K_j$ and a multiset of terms S such that

$$(R_i|_q\phi\downarrow, \tilde{q})\#(S \cup \{R_i|_p\phi\downarrow\}) \in \rho_{\mathsf{io}}(\mathsf{w}_q\phi, \epsilon, \emptyset).$$

For short, $\rho_{\mathsf{io}}(t, \epsilon, \emptyset)$ will be written as $\rho_{\mathsf{io}}(t)$. This can be rewritten using u_p and u_q as:

$$(u_q\sigma_q|_{\tilde{q}}, \tilde{q}) \# (S \cup \{u_p\sigma_p|_{\tilde{p}}\}) \in \rho_{\mathsf{io}}(u_q\sigma_q)$$

Moreover, if $s\#S \in \rho_{\mathsf{io}}(t)$ then $\delta_P(s)\#\delta_P(S) \in \rho_{\mathsf{io}}(\delta_P(t))$ thanks to Lemma 6.5.6, it translates to

$$(\delta_P(u_q\sigma_q|_{\tilde{q}}), \tilde{q}) \#(\delta_P(S) \cup \{\delta_P(u_q\sigma_p|_{\tilde{p}})\}) \in \rho_{\mathsf{io}}(\delta_P(u_q\sigma_q))$$

and as $\delta_P(u|_p) = \delta_P(u)|_p$ for any term u and position p:

$$(\delta_P(u_q\sigma_q)|_{\tilde{q}}, \tilde{q}) \#(\delta_P(S) \cup \{\delta_P(u_q\sigma_p)|_{\tilde{p}}\}) \in \rho_{\mathsf{io}}(\delta_P(u_q\sigma_q))$$

where $\delta_P(S)$ is just $\{\delta_P(s)|s \in S\}$. Because (tr, ϕ) is well-typed, σ_p and σ_q are well-typed substitutions and we get

$$(\delta_P(u_q)|_{\tilde{q}}, \tilde{q}) \#(\delta_P(S) \cup \{\delta_P(u_q)|_{\tilde{p}}\}) \in \rho_{\mathsf{io}}(\delta_P(u_q))$$

which we rewrite to use the substitution-style notation for δ_P, to be:

$$(u_q\delta_P|_{\tilde{q}}, \tilde{q}) \#(\delta_P(S) \cup \{u_q\delta_P|_{\tilde{p}}\}) \in \rho_{\mathsf{io}}(u_q\delta_P).$$

It corresponds to an edge $\alpha_q \rightarrow^{\tilde{p}} \alpha_p$ in the (unrefined) dependency graph of P as $u_q\delta_P|_{\tilde{q}} \in \rho_{\mathsf{out}}(u_q\delta_P)$ and $u_q\delta_P|_{\tilde{p}} \in \rho_{\mathsf{out}}(u_q\delta_P)$ (see Lemma 6.5.5: ϕ does not contain any constant of Σ_0 in key position, deducible keys have then to be nonces, which cannot be honestly typed if deducible). If $\alpha_q \rightarrow^{\tilde{p}} \alpha_p$ is absent from the refined dependency graph, $u_q\delta_P|_{\tilde{q}}$ could be public type, Lemma 6.5.5 ensures then $\phi_0 \vdash u_q\sigma_q|_{\tilde{q}}$, which contradicts the fact that R_i is an asap recipe (as one of its subterms is not asap).

Otherwise, there exists a marked subterm (α_p, p') such that p' is a prefix of $\tilde{p}$. $u_q\sigma_q|_{\tilde{q}}$ was deduced with the recipe $R_i|_q$ and so, because R_i contains only destructors, there exists a prefix q'' of q such that $R_i|_{q''}\phi\downarrow = u_q\sigma_q|_{q'}$. As such, similarly to the previous case and using Definition 6.3.3 for appropriate marked positions, $u_q\sigma_q|_{q'}$ is not deduced with an asap recipe, and neither is $u_q\sigma_q|_q$, which contradicts the asap hypothesis on (tr, ϕ), and let us conclude that $\alpha_q \rightarrow^{\tilde{p}} \alpha_p$. Consequently, for any $p \in \bigcup_{j\geq 1} K_j$, $\alpha_0 \rightarrow^+ \alpha_p$. In particular, by considering the position p such that $\alpha_p = \beta$ and the previous case, be get that $\alpha \rightarrow \alpha_0$ and $\alpha_0 \rightarrow^+ \beta$, which leads to $\alpha \rightarrow^+ \beta$.

To conclude, we finally need to address the case where (tr, ϕ) is only pseudo well-typed, without being well-typed. In that case, there exists a public constant $d \in \Sigma_0$, a substitution $\gamma = \{d \mapsto \langle\omega, \omega\rangle\}$ and a well-typed trace $(\mathsf{tr}', \phi') \in \mathsf{trace}(P)$ such that $\mathsf{tr} = \mathsf{tr}'\gamma$. Because (tr', ϕ') is well-typed, the result proved so far can be applied, and because P is a simple protocol, the sequential dependencies of tr' and tr are identical. As tr and tr' are asap traces, and both d and ω are in the initial knowledge of the attacker, the execution graph of tr and tr' are identical, hence ensuring that the execution graph of (tr, ϕ) is respectful. □

Note that, in the class of simple process, once the trace tr is fixed, the label $\alpha \in \mathcal{L}(P)$ of an action occurring in tr is uniquely defined.

As Proposition 6.5.1 ensures paths from the execution graph of a trace can be mapped to paths of the dependency graph of a protocol, Corollary 6.5.1 enables us to then derive a bound on the depth of such an execution graph, thus providing us the first step to bound the length of the trace itself.

Corollary 6.5.1. Let P be a simple protocol type-compliant w.r.t. some structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and with an acyclic (possibly refined) dependency graph G. Let $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ be an asap and pseudo-well-typed trace w.r.t. $(\mathcal{T}_P, \delta_P)$, and D its corresponding execution graph. We have that:

$$\mathsf{depth}(D) \leq \mathsf{depth}(G) + 1.$$

Proof. Let a and b be actions of tr, $a \to b$ be an edge of D. Let α (resp. β) be the label of action a (resp. b). As P is simple, any action $\mathsf{out}(c, c')$ for two channels c and c' appears without outgoing edge in D. As such the depth of D is the length of a maximal path in D between actions which are not outputs of fresh channels plus one. Suppose now a and b corresponds to actions which are not outputs of fresh channels.

Now, we need to prove that $\alpha \to^+ \beta$. By Definition 6.5.3, this arrow can either correspond to a sequential dependency or a data dependency (for an input). In the first case, by Definition 6.2.2, the same arrow exists in the refined dependency graph if b is not an action $\mathsf{out}(c, c')$ for some channels c and c'; and in the latter, Proposition 6.5.1 ensures also $\alpha \to^+ \beta$. As the relation $\to^+$ if finite with depth $\mathsf{depth}(G)$, we finally get (taking into account the additional edge for the output of a fresh channel):

$$\mathsf{depth}(D) \leq \mathsf{depth}(G) + 1.$$

□

Bounding the width of D

The width of D is the maximal number of outgoing edges of any vertex of D. Actually, any recipe involved in an asap and pseudo-well-typed trace is of the form $C[R_1, \ldots, R_n]$ where C contains only constructors and each R_i contains only destructors. Since messages stored in the frame are well-typed, we can not stack more than $\|\mathsf{out}_P\|$ (maximal size of an outputted term in a well-typed trace) destructors in such a recipe. Note that some key chains may be needed to deduce a message, but the length of such a chain is bounded by $\mathsf{depth}(G)$. Hence, we have that each R_i involves no more than $(1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}$ recipe variables, and we have also that $n \leq \|\mathsf{in}_P\|$ where $\|\mathsf{in}_P\|$ denotes the maximal size of any input term in a well-typed trace. Thus, we have that:

$$\mathsf{width}(D) \leq 1 + (1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1} \times \|\mathsf{in}_P\|.$$

To prove that, Corollary 6.5.2 first aims at bounding the size of any destructor-only asap recipe used in a well-typed witness of equivalence. It relies on the fact that chains of keys that may appear in such a recipe are necessarily bounded thanks to Proposition 6.5.1.

Corollary 6.5.2. Let P be a simple protocol type-compliant w.r.t. a structure-preserving typing system $(\mathcal{T}_P, \delta_P)$ with refined dependency graph G, $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ a pseudo-well-typed trace of P. Then if R is an asap recipe with M leaves of $R\phi\downarrow$ and contains only destructors, then $M \leq (1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}$.

Proof. Uses proof of Proposition 6.5.1. Using the same notations, $M = |\bigcup_{j \in \mathbb{N}} K_i|$. Note that there exists j_0 such that $\forall j > j_0, K_j = \emptyset$, and as such this union is indeed finite. Moreover, for any $j \in \{1, \ldots, j_0\}$ and $p \in K_j$, there exists $q \in K_{j-1}$ such that $\alpha_q \to^{\tilde{p}} \alpha_q$. As such, the labels of the elements of $\bigcup_{j \in \mathbb{N}} K_j$ form a subtree of G

of root α_0, and of depth lesser or equal than $\mathsf{depth}(G)$. The degree of this subtree is bounded by the number of constructors (and thus the number of encryption layers) in any term of a pseudo-well-typed frame, which is it self bounded by $\|\mathsf{out}_P\|$. Hence its degree is at most $\|\mathsf{out}_P\|$, leading to a subtree of size at most $\sum_{k=0}^{\mathsf{depth}(G)} \|\mathsf{out}_P\|^k$ and thus $M \leq (1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}$. □

Having bounded the size of recipes used in our witness of non-equivalence, we can translate it into a bound on the width of the execution graph of this witness, as the maximal size of recipes corresponds to the maximal arity of such a graph; thus providing us with the second step towards the final bound on the length of a minimal witness of non-equivalence.

Lemma 6.5.7. Let P and Q be two simple protocol type-compliant w.r.t. their respective structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$; and $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ a witness of $P \not\approx' Q$ (*i.e.* the witness is a trace of P without necessarily being a trace of Q) with execution graph D. Then

$$\mathsf{width}(D) \leq 1 + (1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1} \times \|\mathsf{in}_P\|$$

Proof. The bound comes from Corollary 6.5.2: any term can have at most one sequential predecessor and any asap input recipe of tr is of the form $C[R_1, \ldots, R_n]$ where C contains only constructors and for any i, R_i has M_i leaves and contains only destructors and as such $M_i \leq \|\mathsf{out}_P\|^{\mathsf{depth}(G)}$. Because (tr, ϕ) is pseudo-well-typed, each input from the protocol has at most $\|\mathsf{in}_P\|$ leaves. Hence any action of D can have at most 1 predecessor (for the sequential dependency) and $\|\mathsf{out}_P\|^{\mathsf{depth}(G)} \times \|\mathsf{in}_P\|$ other predecessors for the data dependencies, leading to the desired result. □

6.5.3 Bounding the length of a minimal witness

To conclude, we need to bound the length of a minimal witness of non-equivalence. We have already seen that we can consider a witness that is asap and pseudo-well-typed, and we have bound the depth and the width of its associated dag. Still, the length of such a trace may be arbitrary long. We now show that we can bound the number of roots (vertices with no ingoing edge) in a minimal witness of non-equivalence.

To do so, we introduce the notion of pruning of a graph w.r.t. some vertices. It intuitively corresponds, for an execution graph, to the retrieving of the causal dependencies required to fire some action.

Definition 6.5.4 (pruning)**.** Given an execution graph $D = (V, E)$ and a set R of roots of D, we define the *pruning* $D_R = (V_R, E_R)$ of D w.r.t. R as follows:

- $V_R = \{v \in V | \exists r \in R, r \rightarrow^* v\}$
- $E_R = \{(u, v) \in E | u, v \in V_R\}$

where $\rightarrow^*$ denotes the transitive closure of the relation induced by E.

Lemma 6.5.8 (properties of pruning)**.** Let P be a simple protocol, $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and D the execution graph of tr w.r.t. P. Let $R = \{v_1, \ldots, v_p\}$ be a set of nodes of D and D_R the pruning of D w.r.t. R. Then there exits $(\mathsf{tr}_R, \phi_R) \in \mathsf{trace}(P)$ such that:

- D_R is the execution graph of tr_R w.r.t. P,
- tr_R is a subtrace of tr and ϕ_R is a subframe of ϕ,

Proof. Intuitively, the execution graph captures all the dependencies to execute any vertex/action in the corresponding trace. The closure of D_R ensures that any action occurring in tr_R has the needed predecessors. □

Lemma 6.5.9. Let P and Q be two simple protocols type-compliant w.r.t. some structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$, and such that $P \not\sqsubseteq Q$. Let (tr, ϕ) be a witness of non-inclusion which is asap, pseudo-well-typed and with minimal length, and D be its corresponding execution graph. We have that:

$$\mathsf{nbroot}(D) \leq 2 \times (1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}.$$

Proof. There are two main reasons of non inclusion.

- Either Q is not able to mimic the last action of the trace tr. In that case, we prune D by selecting only the last action of tr and its (successive) sons.
- Or the resulting frames are not in static equivalence. Consider an equality test $R_1 = R_2$ that witnesses non static equivalence. We show that R_1 and R_2 can be chosen to be destructor-only, and we bound the number of recipe variables involved in R_1 and R_2 by $(1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}$. Thus, in this case, we prune D by selecting the actions producing these variables and their successive sons.

More formally, (tr, ϕ) can be of two forms:

- If there exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ and $\phi \not\sim \psi$. From Lemma A.1.13, any meaningful test to distinguish between two frames is always made of two destructor-only recipes R and R' such that, for instance, $R\phi\downarrow = R'\phi\downarrow$ while $R\psi\downarrow \neq R'\psi\downarrow$. Thus only twice as many variables as the number of leaves in such a recipe are needed in the frames to distinguish between them. Such recipes have at most $(1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}$ leaves as per Corollary 6.5.2. Thus by selecting those $2 \times (1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}$ variables (as there are two recipes) in ϕ and pruning D w.r.t. these, we obtain (tr_0, ϕ_0) which is both a trace of P and Q (by Lemma 6.5.8) and such that ϕ_0 (resp ψ_0) is a subframe of ϕ (resp. ψ) such that $dom(R) \cup dom(R') \subseteq dom(\phi_0)$ (resp. $dom(R) \cup dom(R') \subseteq dom(\psi_0)$). Thus $R\phi\downarrow = R\phi_0\downarrow$, $R'\phi\downarrow = R'\phi_0\downarrow$, $R\psi\downarrow = R\psi_0\downarrow$ and $R'\psi\downarrow = R'\psi_0\downarrow$, leading to $R\phi_0\downarrow = R'\phi_0\downarrow$ and $R\psi_0\downarrow \neq R'\psi_0\downarrow$, and witnessing $\phi_0 \not\sim \psi_0$.
- Or $(\mathsf{tr}, \psi) \notin \mathsf{trace}(Q)$ for any ψ. There exists an action a of tr which cannot be executed in Q. Let us prune tr w.r.t. this single action. We obtain (tr_0, ϕ_0) which is a trace of P (by Lemma 6.5.8) and such that for any ψ_0, $(\mathsf{tr}, \psi_0) \notin \mathsf{trace}(Q)$ as Lemma 6.5.8 ensures the execution steps of tr_0 through are unchanged compared to the corresponding actions in tr. Consequently, in this case, we need only to consider only one root.

Putting the two bounds together, we need at most $2\times(1+\|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}$ roots in D to get a witness of $P \not\sqsubseteq Q$. □

In conclusion, we have bound the (minimal) length of a witness of non equivalence. This, in turn, bounds the number of sessions. We then conclude using *e.g.* Chapter 5 since trace equivalence is decidable for a bounded number of sessions. Since trace equivalence is NP for a bounded number of sessions [12], we deduce decidability in triple exponential time, in the size of the protocols.

Theorem 6.3.1. The problem of deciding whether two simple protocols P and Q, type-compliant w.r.t. some structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$, and with acyclic refined dependency graphs obtained relying on appropriate markings $\mathcal{M}_P$ and $\mathcal{M}_Q$ are trace equivalence (*i.e.* $P \approx Q$) is decidable.

Proof. Suppose $P \not\approx Q$. By Lemma 6.5.2, $P \not\approx' Q$. Two cases occur, depending on whether the witness of non-equivalence is a trace of P or Q (see Definition 6.5.2). In the first case, as the witness is asap and pseudo-well-typed, we can apply Corollary 6.5.1, Lemma 6.5.7 and Lemma 6.5.9 to respectively obtain a bound on the depth, the width and the number of roots for the execution graph corresponding to this witness. Let G_P (resp. G_Q) be the (refined) dependency graph of P (resp. G_Q). Thus, we get a bound on the length A of such a minimal witness:

$$2(1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G_P)+1}(1 + \|\mathsf{in}_P\|(1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G_P)+1})^{\mathsf{depth}(G_P)+1}.$$

The second case is handled symmetrically, leading to a bound B on the size of a witness in $\mathsf{trace}(Q)$:

$$2(1 + \|\mathsf{out}_Q\|)^{\mathsf{depth}(G_Q)+1}(1 + \|\mathsf{in}_Q\|(1 + \|\mathsf{out}_Q\|)^{\mathsf{depth}(G_Q)+1})^{\mathsf{depth}(G_Q)+1}.$$

Hence a witness of $P \not\approx Q$ has length at most $\max(A, B)$. We conclude by invoking a decidability result for a bounded number of sessions such as Theorem 5.1.1. □

6.6 Conclusion

We have obtained the first decidability result for trace equivalence, for an unbounded number of sessions and unrestricted nonces.

Generating a structure-preserving typing system (actually the more fine-grained one) for which type-compliance is satisfied, and checking acyclicity of the resulting dependency graph is not difficult but rather cumbersome. We plan to devise a script to perform these steps automatically. We also plan to study how to relax some of our assumptions. First, we think that the "simple protocols" assumption could be relaxed to consider action-determinate protocols. Second, we plan to investigate other criteria to soundly remove edges in the dependency graph, in order to get rid of meaningless cycles. Lastly, our result applies only to protocols with concatenation and symmetric encryption. We inherit this restriction from Chapter 4. We believe that once the typing result of Chapter 4 is extended to all standard primitives then our decidability result will extend as well.

The current complexity of our result is too high to use existing tools that decide trace equivalence for a bounded number of sessions (they typically handle up to 2-3 sessions). However, since our result bounds quite precisely the form of a minimal attack, it seems possible to improve its complexity and to use model-checkers instead.

Our decidability result intuitively encompasses decidability of secrecy, expressed as a trace property, since secrecy can be encoded using trace equivalence. We believe that our proof technique could be applied to decide authentication properties as well, for which we are not aware of any decidability result. The main difficulty induced by authentication properties is that authentication implicitly introduces disequalities (there might be an attack because agent B received a message *different* from the one sent by agent A). However, deciding trace equivalence also requires a careful treatment of disequalities. As future work, we plan to formally apply our technique to obtain decidability of a fragment of trace properties that encompasses secrecy and authentication.

Chapter 7

Decidability of trace equivalence for ping-pong protocols

In this chapter, we study the decidability of a class of protocols for which trace equivalence is decidable. As secrecy is already undecidable in general, we therefore focus on a class of protocols for which secrecy is decidable [31]. This class, called *ping-pong protocols*, typically assumes that each protocol rule manipulates at most one variable and that the protocol is formed of a set of independent in/out rules. Intuitively, this corresponds to the assumption that, at each step of the protocol, upon receiving a message there is at most one part of it that is unknown to the agent (typically a key, a nonce, or an encrypted packet).

Surprisingly, while this class is decidable for reachability, even a fragment of it (with only symmetric encryption) turns out to be undecidable for equivalence properties. We consequently further assume our protocols to be deterministic (that is, given an input, there is at most one possible output). We show that equivalence is decidable for an unbounded number of sessions and for protocols with randomised symmetric and asymmetric encryption, and signatures. Since we need to assume our constructors to be randomised and since we assume "at most one variable", we can only handle a very limited notion of (randomised) concatenation that appends atomic values.

Interestingly, we show that checking for equivalence of protocols actually amounts into checking equality of languages of deterministic pushdown automata. The decidability of equality of languages of deterministic pushdown automata is a difficult problem, shown to be decidable [62]. We actually characterise equivalence of protocols in terms of equivalence of deterministic generalised real-time pushdown automata, that is deterministic pushdown automata with no epsilon-transition but such that the automata may unstack several symbols at a time. More precisely, we show how to associate to a process P an automata $\mathcal{A}_P$ such that two processes are equivalent if, and only if, their corresponding automata yield the same language and, reciprocally, we show how to associate to an automata $\mathcal{A}$ a process $P_{\mathcal{A}}$ such that two automata yield the same language if, and only if, their corresponding processes are equivalent, that is:

$$P \approx Q \Leftrightarrow L(\mathcal{A}_P) = L(\mathcal{A}_Q) \quad \text{and} \quad L(\mathcal{A}) = L(\mathcal{B}) \Leftrightarrow P_{\mathcal{A}} \approx P_{\mathcal{B}}.$$

Therefore, checking for equivalence of protocols is as difficult as checking equivalence of deterministic generalised real-time pushdown automata.

To transform equivalence of processes into equivalence of pushdown automata, we first show how to get rid of an active attacker. More precisely, we show that

$$P \approx Q \Leftrightarrow P' \approx_{\mathsf{fwd}} Q'$$

where $\approx_{\mathsf{fwd}}$ intuitively represents equivalence of processes when the attacker may only *forward* messages. This equivalence is obtained by partially encoding the attacker in P' and Q', still preserving equivalence.

The decision procedure for checking equivalence of deterministic pushdown automata has been recently implemented by G. Sénizergues [50]. We have therefore implemented our transformation from processes to pushdown automata, yielding the first tool that decides equivalence of (some class of) protocols for an unbounded number of sessions. As an application, we have analysed several protocols of the literature, including a simplified version of the basic access control protocol (BAC) of the biometric passport [52].

We characterise the notion of ping-pong protocols in Section 7.1 and state our main results. Sections 7.2 and 7.3 are devoted to decidability. More precisely, we show in Section 7.2 how to get rid of an active attacker by encoding it directly in the process. Next, we show in Section 7.3 how to encode equivalence between processes (in presence of a forwarder attacker) into equivalence of pushdown automata, characterising further which cases may result in non equivalence. Finally, we study in Section 7.4 the converse translation and show that equivalence of pushdown automata can be reduced to equivalence of protocols. We present our implementation and its application to protocols in Section 7.5. Concluding remarks can be found in Section 7.6.

7.1 Ping-pong protocols

We aim at providing a decidability result for the problem of trace equivalence between protocols in presence of replication. However, it is well-known that replication leads to undecidability even for the simple case of reachability properties. Thus, we consider a class of protocols, called $\mathcal{C}_{\mathsf{pp}}$, for which (in a slightly different setting), reachability has already been proved decidable [31].

7.1.1 Term algebra

In order to express the results of the chapter, we introduce a variation of our term algebra, compared to one detailed in Chapter 2. In particular, we consider a signature which can be seen as an instance of our original one, with *sorts* attached to terms. Moreover, we do not consider pairing, and focus on randomised encryption with atomic keys. Other definitions, such as the process algebra and semantics, stay as described in Chapter 2.

As usual, messages are represented by terms. More specifically, we consider a *sorted signature* with six sorts rand, key, msg, SymKey, PrivKey and PubKey that represent respectively random numbers, keys, messages, symmetric keys, private keys and public keys. We assume that msg subsumes the five other sorts, key subsumes SymKey, PrivKey and PubKey. We consider six function symbols senc and sdec, raenc and radec, sign and check that represent symmetric, asymmetric encryption and decryption as well as signatures. Since we are interested in the analysis of indistinguishability properties, we consider a randomised encryption scheme:

$$\begin{array}{rlrrlr} \mathsf{senc}: & \mathsf{msg}\times\mathsf{SymKey}\times\mathsf{rand} & \to \mathsf{msg} & \mathsf{sdec}: & \mathsf{msg}\times\mathsf{SymKey} & \to \mathsf{msg} \\ \mathsf{raenc}: & \mathsf{msg}\times\mathsf{PubKey}\times\mathsf{rand} & \to \mathsf{msg} & \mathsf{radec}: & \mathsf{msg}\times\mathsf{PrivKey} & \to \mathsf{msg} \\ \mathsf{sign}: & \mathsf{msg}\times\mathsf{PrivKey}\times\mathsf{rand} & \to \mathsf{msg} & \mathsf{check}: & \mathsf{msg}\times\mathsf{PubKey} & \to \mathsf{msg} \end{array}$$

We discuss in Section 7.5 how we can handle a limited notion of (randomised) concatenation.

We further assume an infinite set Σ_0 of *constant symbols* of sort key or msg, an infinite set $\mathcal{C}h$ of constant symbols of sort channel, two infinite sets of *variables* $\mathcal{X}, \mathcal{W}$, and an infinite set of *names* $\mathcal{N} = \mathcal{N}_{\mathsf{pub}} \uplus \mathcal{N}_{\mathsf{priv}}$ of *names* of sort rand: $\mathcal{N}_{\mathsf{pub}}$ represents the random numbers drawn by the attacker while $\mathcal{N}_{\mathsf{priv}}$ represents the random numbers drawn by the protocol's participants.

As usual, *terms* are defined as names, variables, and function symbols applied to other terms. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{N}, \mathcal{X})$ the set of terms built on function symbols in $\mathcal{F}$, names in $\mathcal{N}$, and variables in $\mathcal{X}$. We simply write $\mathcal{T}(\mathcal{F}, \mathcal{N})$ when $\mathcal{X} = \emptyset$. We consider three particular signatures:

$$\Sigma_{\mathsf{pub}} = \{\mathsf{senc}, \mathsf{sdec}, \mathsf{raenc}, \mathsf{radec}, \mathsf{sign}, \mathsf{check}, \mathsf{start}\}$$

$$\Sigma^+ = \Sigma_{\mathsf{pub}} \cup \Sigma_0 \qquad \Sigma = \{\mathsf{senc}, \mathsf{raenc}, \mathsf{sign}, \mathsf{start}\} \cup \Sigma_0$$

where $\mathsf{start} \notin \Sigma_0$ is a constant symbol of sort msg. The signature Σ_{pub} represents the functions/data available to the attacker, including a constant start used to start sessions of the protocols. The signature Σ^+ is the most general signature, while Σ models actual messages (with no failed computation). We assume a bijection between elements of sort PrivKey and PubKey. If k is a constant of sort PrivKey, k^{-1} will denotes its image by this function, called *inverse*. The inverse of the inverse function is also denoted by $\_^{-1}$, so that $(\mathsf{k}^{-1})^{-1} = \mathsf{k}$. To keep homogeneous notations, we extend this function to symmetric keys: if k is of sort SymKey, then $\mathsf{k}^{-1} = \mathsf{k}$. The relation between encryption and decryption is represented through the following rewriting rules, yielding a convergent rewrite system:

$$\mathsf{sdec}(\mathsf{senc}(x, \mathsf{k}_1, z), \mathsf{k}_1) \to x$$
$$\mathsf{radec}(\mathsf{raenc}(x, \mathsf{k}_2, z), {\mathsf{k}_2}^{-1}) \to x \qquad \mathsf{check}(\mathsf{sign}(x, \mathsf{k}_3, z), {\mathsf{k}_3}^{-1}) \to x$$

with k_1 of sort SymKey, k_2 of sort PubKey, and k_3 of sort PrivKey. For instance, the first rule models the fact that the decryption of a ciphertext will return the associated plaintext when the right key is used to perform decryption. The two last rules are used to model asymmetric encryption and signatures. We denote by $t{\downarrow}$ the *normal form* of a term $t \in \mathcal{T}(\Sigma^+, \mathcal{N}, \mathcal{X})$.

Example 7.1.1. We consider a simplified version of the protocol presented in [41]. The purpose of this protocol informally described below is to establish a key k_{AB} between two participants A and B using public key encryption and signature.

$$1. \quad A \to B : \quad \mathsf{raenc}(\mathsf{sign}(\mathsf{k}_{AB}, \mathsf{sk}_A, r_A^1), \mathsf{pk}_B, r_A^2)$$
$$2. \quad B \to A : \quad \mathsf{ack}$$

The agent A sends a symmetric key k_{AB} signed with A's private key sk_A (using a fresh random number r_A^1), and the resulting ciphertext is encrypted with B's public key pk_B (using a fresh random number r_A^2). The agent B answers to this request by decrypting this message, and verifying the signature. If all checks succeed, B informs the agent A by sending an acknowledgement, *i.e.* the constant ack. The agents A and B can now use the symmetric key k_{AB} to communicate.

The role of agent A is modelled by a process P_A while the role of agent B is modelled by P_B. We have that:

$$P_A \stackrel{\mathsf{def}}{=} \; !\,\mathsf{in}(c_A, \mathsf{start}).\mathsf{new}\, r_A^1.\mathsf{new}\, r_A^2.\mathsf{out}(c_A, \mathsf{raenc}(\mathsf{sign}(\mathsf{k}_{AB}, \mathsf{sk}_A, r_A^1), \mathsf{pk}_B, r_A^2)) \quad (1)$$
$$\mid !\,\mathsf{in}(c'_A, \mathsf{start}).\mathsf{new}\, r_A^1.\mathsf{new}\, r_A^2.\mathsf{out}(c'_A, \mathsf{raenc}(\mathsf{sign}(\mathsf{k}_{AC}, \mathsf{sk}_A, r_A^1), \mathsf{pk}_C, r_A^2)) \quad (2)$$
$$P_B \stackrel{\mathsf{def}}{=} \; !\,\mathsf{in}(c_B, \mathsf{raenc}(\mathsf{sign}(x, \mathsf{sk}_A, z_1), \mathsf{pk}_B, z_2)).\mathsf{out}(c_B, \mathsf{ack}) \quad (3)$$

The constants c_A, c'_A and c_B are constants of sort channel, ack is a constant of sort msg, whereas the constants $\mathsf{k}_{AB}, \mathsf{k}_{AC}, \mathsf{sk}_A, \mathsf{sk}_B, \mathsf{sk}_C, \mathsf{pk}_A, \mathsf{pk}_B, \mathsf{pk}_C$ which are in Σ_0 are such that:

- $\mathsf{k}_{AB}, \mathsf{k}_{AC}$ are of sort SymKey,
- $\mathsf{sk}_A, \mathsf{sk}_B, \mathsf{sk}_C$ are of sort PrivKey, and
- $\mathsf{pk}_A, \mathsf{pk}_B, \mathsf{pk}_C$ of sort PubKey.

Moreover, we have that $\mathsf{sk}_X^{-1} = \mathsf{pk}_X$ for $X \in \{A, B, C\}$ whereas $\mathsf{k}_{AB}^{-1} = \mathsf{k}_{AB}$ and $\mathsf{k}_{AC}^{-1} = \mathsf{k}_{AC}$. Finally, r_A^1, r_A^2 are names of sort rand, and x (resp. z_1, z_2) is a variable of sort msg (resp. rand).

Intuitively, P_A sends k_{AB} signed with sk_A and encrypted with pk_B to the agent B (branch 1). More generally, the agent A can start different sessions with different agents. Thus, the process P_A models the agent A initiating a session with B (branch 1) as well as with C (branch 2). The process P_B models the agent B answering a request from A. We could also consider the scenario where the agent B is also willing to talk to C or where the initiator, here played by A, is also played by other agents such as B. We consider here only a simpler case to keep the example reasonably short.

To model the whole protocol, we sent the public key $\mathsf{pk}_A, \mathsf{pk}_B, \mathsf{pk}_C$ in clear, as well as the private key sk_C, to model the fact that the attacker may learn the private keys of some corrupted agents. This is modelled through the following process P_{key}:

$$\begin{array}{rcl} P_{\mathsf{key}} & \stackrel{\mathsf{def}}{=} & !\,\mathsf{in}(c_1,\mathsf{start}).\mathsf{out}(c_1,\mathsf{pk}_A) \mid !\,\mathsf{in}(c_2,\mathsf{start}).\mathsf{out}(c_1,\mathsf{pk}_B) \mid \\ & & !\,\mathsf{in}(c_3,\mathsf{start}).\mathsf{out}(c_3,\mathsf{pk}_C) \mid !\,\mathsf{in}(c_4,\mathsf{start}).\mathsf{out}(c_4,\mathsf{sk}_C) \end{array}$$

Then, the whole protocol is given by P, where P_A, P_B, and P_{key} evolve in parallel:

$$P \stackrel{\mathsf{def}}{=} P_A \mid P_B \mid P_{\mathsf{key}}$$

This protocol is actually insecure as demonstrated by the following attack:

$$\begin{array}{lll} 1. & A \to C: & \mathsf{raenc}(\mathsf{sign}(\mathsf{k}_{AC},\mathsf{sk}_A,r_A^1),\mathsf{pk}_C,r_A^2) \\ 2. & C(A) \to B: & \mathsf{raenc}(\mathsf{sign}(\mathsf{k}_{AC},\mathsf{sk}_A,r_A^1),\mathsf{pk}_B,r_C^1) \\ 3. & B \to A: & \mathsf{ack} \end{array}$$

A initiates a session with a malicious user C sending him a key k_{AC}. This malicious user then legally learns k_{AC} but also its signature $\mathsf{sign}(\mathsf{k}_{AC},\mathsf{sk}_A,r_A^1)$ under the signing key of A. He may then resend this key to B in the name of A. The agent B accepts the key k_{AC} as being a secret key between A and B.

This attack can be formalised in our model as follows:

1. The public keys of all the participants are disclosed as well as the secret key sk_C of the corrupted agent C. Formally, let $K_0 \stackrel{\mathsf{def}}{=} (P;\emptyset)$, we have that:

$$K_0 \xrightarrow{\mathsf{in}(c_1,\mathsf{start}).\mathsf{out}(c_1,\mathsf{w}_1).\mathsf{in}(c_2,\mathsf{start}).\mathsf{out}(c_2,\mathsf{w}_2).\mathsf{in}(c_3,\mathsf{start}).\mathsf{out}(c_3,\mathsf{w}_3)} \xrightarrow{\mathsf{in}(c_4,\mathsf{start}).\mathsf{out}(c_4,\mathsf{w}_4)} (P;\sigma_0)$$

   where $\sigma_0 = \{\mathsf{w}_1 \triangleright \mathsf{pk}_A,\ \mathsf{w}_2 \triangleright \mathsf{pk}_B,\ \mathsf{w}_3 \triangleright \mathsf{pk}_C,\ \mathsf{w}_4 \triangleright \mathsf{sk}_C\}$.

2. The agent A initiates a session with C and sends the corresponding encrypted message. More formally, we have that:

$$(P;\sigma_0) \xrightarrow{\mathsf{in}(c'_A,\mathsf{start}).\mathsf{out}(c'_A,\mathsf{w}_5)} (P;\sigma)$$

   where $\sigma = \sigma_0 \cup \{\mathsf{w}_5 \triangleright \mathsf{raenc}(\mathsf{sign}(\mathsf{k}_{AC},\mathsf{sk}_A,r_A^1),\mathsf{pk}_C,r_A^2)\}$ and r_A^1, r_A^2 are (fresh) names in $\mathcal{N}_{\mathsf{priv}}$.

Hence, we have that $(\mathsf{tr},\sigma) \in \mathsf{trace}(K_0)$ where:

$$\begin{array}{rl} \mathsf{tr} = & \mathsf{in}(c_1,\mathsf{start}).\mathsf{out}(c_1,\mathsf{w}_1).\mathsf{in}(c_2,\mathsf{start}).\mathsf{out}(c_2,\mathsf{w}_2).\mathsf{in}(c_3,\mathsf{start}).\mathsf{out}(c_3,\mathsf{w}_3). \\ & \mathsf{in}(c_4,\mathsf{start}).\mathsf{out}(c_4,\mathsf{w}_4).\mathsf{in}(c'_A,\mathsf{start}).\mathsf{out}(c'_A,\mathsf{w}_5). \end{array}$$

In this execution trace, first the keys $\mathsf{pk}_A, \mathsf{pk}_B, \mathsf{pk}_C$ and sk_C are sent after having called the corresponding process. Then, branch (2) of P is triggered. Going further, our naive protocol is secure if the key received by B remains private. To model this, we modify the process P_B as follows:

$$\begin{array}{rcl} P_B^l & \stackrel{\mathsf{def}}{=} & !\,\mathsf{in}(c_B,\mathsf{raenc}(\mathsf{sign}(x,\mathsf{sk}_A,z_1),\mathsf{pk}_B,z_2)).\mathsf{out}(c_B,x) \\ P_B^r & \stackrel{\mathsf{def}}{=} & !\,\mathsf{in}(c_B,\mathsf{raenc}(\mathsf{sign}(x,\mathsf{sk}_A,z_1),\mathsf{pk}_B,z_2)).\mathsf{out}(c_B,\mathsf{k}) \end{array}$$

Then, to model secrecy of the key received by B, we consider the following equivalence: $P_A \mid P_B^l \mid P_{\mathsf{key}} \approx P_A \mid P_B^r \mid P_{\mathsf{key}}$. An attacker should not distinguish between two instances of the protocol, one where B used the key established through the protocol and one where a magic key k is used instead.

However, our protocol is insecure. An attacker may easily learn k_{AC}, and sends to B a message of the expected form (as if it was issued by A) and that will contain this corrupted key instead of k_{AB}. Formally, we have that:

$$P_A \mid P_B^l \mid P_{\mathsf{key}} \not\approx P_A \mid P_B^r \mid P_{\mathsf{key}}.$$

This is reflected by the trace tr' described below:

$$\mathsf{tr}' \stackrel{\mathsf{def}}{=} \mathsf{tr}.\mathsf{in}(c_B, \mathsf{raenc}(\mathsf{radec}(\mathsf{w}_5, \mathsf{sk}_C), \mathsf{w}_2, r_C)).\mathsf{out}(c_B, \mathsf{w}_6)$$

where r_C is a name in $\mathcal{N}_{\mathsf{pub}}$.

We have that $(\mathsf{tr}', \sigma_1) \in \mathsf{trace}(K_0)$ with $K_0 = (P_A \mid P_B^l \mid P_{\mathsf{key}}; \sigma_1)$ where σ_1 is defined below. Because of the existence of only one branch using each channel, there is only one possible execution of $P_A \mid P_B^r \mid P_{\mathsf{key}}$ (up to a bijective renaming of the private names of sort rand) matching the labels in tr', and the corresponding execution will allow us to reach the frame σ_2 as described below:

1. $\sigma_1 \stackrel{\mathsf{def}}{=} \sigma_0 \cup \{\mathsf{w}_5 \triangleright \mathsf{raenc}(\mathsf{sign}(\mathsf{k}_{AC}, \mathsf{sk}_A, r_A^1), \mathsf{pk}_C, r_A^2),\ \mathsf{w}_6 \triangleright \mathsf{k}_{AC}\}$,
2. $\sigma_2 \stackrel{\mathsf{def}}{=} \sigma_0 \cup \{\mathsf{w}_5 \triangleright \mathsf{raenc}(\mathsf{sign}(\mathsf{k}_{AC}, \mathsf{sk}_A, r_A^1), \mathsf{pk}_C, r_A^2),\ \mathsf{w}_6 \triangleright \mathsf{k}\}$.

where k is a (private) constant in Σ_0. We have that $\sigma_1 \not\sim \sigma_2$. Indeed, consider the recipes $R_1 = \mathsf{check}(\mathsf{radec}(\mathsf{w}_5, \mathsf{w}_4), \mathsf{w}_1)$ and $R_2 = \mathsf{w}_6$. We have that $R_1\sigma_1\downarrow = R_2\sigma_1\downarrow = \mathsf{k}_{AC}$, whereas $R_1\sigma_2\downarrow = \mathsf{k}_{AC}$ and $R_2\sigma_2\downarrow = \mathsf{k}$ thus $R_1\sigma_2\downarrow \neq R_2\sigma_2\downarrow$. Hence $\sigma_1 \not\sim \sigma_2$ and $P_A \mid P_B^l \mid P_{\mathsf{key}} \not\approx P_A \mid P_B^r \mid P_{\mathsf{key}}$.

7.1.2 Class $\mathcal{C}_{\mathsf{pp}}$

We basically consider ping-pong protocols (an output is computed using only the message previously received in input), and we assume a kind of determinism. Moreover, we restrict the terms that are manipulated throughout the protocols: only one unknown message (modelled by the use of a variable of sort msg) can be received at each step.

We fix a variable $x \in \mathcal{X}$ of sort msg. An *input term* (resp. *output term*) is a term defined by the grammars given below:

$$u := x \mid \mathsf{s} \mid \mathsf{f}(u, \mathsf{k}, z) \qquad\qquad v := x \mid \mathsf{s} \mid \mathsf{f}(v, \mathsf{k}, r)$$

where $\mathsf{s}, \mathsf{k} \in \Sigma_0 \cup \{\mathsf{start}\}$, $z \in \mathcal{X}$, $\mathsf{f} \in \{\mathsf{senc}, \mathsf{raenc}, \mathsf{sign}\}$ and $r \in \mathcal{N}$. Intuitively, no destructor should be used explicitly. Moreover, we assume that each variable (resp. name) occurs at most once in u (resp. v).

Definition 7.1.1. $\mathcal{C}_{\mathsf{pp}}$ is the class of protocol of the form:

$$P = \bigmid_{i=1}^{n} \bigmid_{j=1}^{p_i} !\mathsf{in}(c_i, u_i^j).\mathsf{new}\ r_1.\ldots.\mathsf{new}\ r_{k_j^i}.\mathsf{out}(c_i, v_i^j)$$

such that:

1. for all $i \in \{1, \ldots, n\}$, and $j \in \{1, \ldots, p_i\}$, $k_i^j \in \mathbb{N}$, u_i^j is an input term, and v_i^j is an output term where names occurring in v_i^j are included in $\{r_1, \ldots, r_{k_i^j}\}$;
2. for all $i \in \{1, \ldots, n\}$, and $j_1, j_2 \in \{1, \ldots, p_i\}$, if $j_1 \neq j_2$ then for any renaming of variables, $u_i^{j_1}$ and $u_i^{j_2}$ are not unifiable[1].

Each subprocess $\mathsf{in}(c_i, u_i^j).\mathsf{new}\ r_1.\ldots.\mathsf{new}\ r_{k_j^i}.\mathsf{out}(c_i, v_i^j)$ is called a *branch* of P.

[1] i.e. there does not exist θ such that $u_i^{j_1}\theta = u_i^{j_2}\theta$.

Item *1* holds for any process representing a protocol: the variables of the output should be bound by the input. Item *2* enforces a deterministic behaviour: a particular input action can only be accepted by one branch of the protocol. This is a natural restriction since most of the protocols are indeed deterministic: an agent should usually know exactly what to do once he has received a message. Actually, the main limitations of the class $\mathcal{C}_{\mathsf{pp}}$ is that we consider a restricted signature (e.g. no pair, no hash function), and names can only be used to produce randomised ciphertexts.

Example 7.1.2. The protocols described in Example 7.1.1 are in $\mathcal{C}_{\mathsf{pp}}$. For instance, we can check that:

- $\mathsf{raenc}(\mathsf{sign}(x, \mathsf{sk}_A, z_1), \mathsf{pk}_B, z_2)$ is an input term, and
- $\mathsf{raenc}(\mathsf{sign}(\mathsf{k}_{AB}, \mathsf{sk}_A, r_A^1), \mathsf{pk}_B, r_A^2)$ is an output term.

Moreover, the determinism condition (item *2*) is clearly satisfied: each branch of the protocol $P_A \mid P_B^l \mid P_{\mathsf{key}}$ (resp. $P_A \mid P_B^r \mid P_{\mathsf{key}}$) uses a different channel.

When studying trace equivalence (or even trace inclusion) we can even safely force a process to perform an input action followed directly by its associated output action.

We consider a set of "big-step" traces, defined as follows.

$$\mathsf{trace}^{\mathsf{io}*}(K) \quad = \quad \left\{ \; (\mathsf{tr}, \sigma) \;\middle|\; \begin{array}{l} K \stackrel{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \sigma) \text{ for some configuration } (\mathcal{P}; \sigma) \\ \text{with } \mathsf{tr} \text{ sequence of input-output blocks.} \end{array} \right\}$$

The notion of trace inclusion (resp. trace equivalence) w.r.t. big-step traces is defined accordingly.

Definition 7.1.2. Let P and Q be two protocols. We have that $P \sqsubseteq^{\mathsf{io}*} Q$ if for every $(\mathsf{tr}, \sigma) \in \mathsf{trace}^{\mathsf{io}*}(P)$, there exists $(\mathsf{tr}', \sigma') \in \mathsf{trace}^{\mathsf{io}*}(Q)$ such that $\mathsf{tr} = \mathsf{tr}'$ and $\sigma \sim \sigma'$. They are *trace equivalent*, written $P \approx^{\mathsf{io}*} Q$, if $P \sqsubseteq^{\mathsf{io}*} Q$ and $Q \sqsubseteq^{\mathsf{io}*} P$.

Due to the form of protocols in $\mathcal{C}_{\mathsf{pp}}$, any trace made up of inputs and outputs actions can be first completed with all the available output actions, and then be mapped to a trace that is made up of input-output blocks only. Thus, we have that the two notions of trace equivalence coincide.

Proposition 7.1.1. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$. We have that $P \sqsubseteq^{\mathsf{io}*} Q$ if, and only if, $P \sqsubseteq Q$.

This proposition easily follows from that fact that for any process of $\mathcal{C}_{\mathsf{pp}}$, any input is immediately followed by an output.

7.1.3 Main results

Our first main contribution is a decision procedure for trace equivalence of processes in $\mathcal{C}_{\mathsf{pp}}$.

Theorem 7.1.1. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$. The problem whether P and Q are trace equivalent, i.e. $P \approx Q$, is decidable.

Deciding trace equivalence is done in two main steps.

1. First, we show how to reduce trace equivalence between protocols in $\mathcal{C}_{\mathsf{pp}}$, to the problem of deciding trace equivalence (still between protocols in $\mathcal{C}_{\mathsf{pp}}$) when the attacker acts as a *forwarder*, that is, when the attacker may only forward messages obtained through the protocol. This step is detailed in Section 7.2.
2. Then, we encode the problem of deciding trace equivalence for forwarding attackers into the problem of language equivalence for real-time generalised pushdown deterministic automata (GPDA), that is, deterministic pushdown automata with no epsilon-transition but such that the automata may unstack several symbols at a time. This step is detailed in Section 7.3

We also provide an implementation of our translation from protocols to pushdown automata, yielding a tool for automatically checking equivalence of security protocols, for an unbounded number of sessions. This contribution is described in Section 7.5.

Actually, we characterise equivalence of protocols in terms of equivalence of GPDA. Indeed, Step (2) above shows how to associate to a process P an automata $\mathcal{A}_P$ such that two processes are equivalent if, and only if, their corresponding automata yield the same language. Conversely, we also show how to associate to an automata $\mathcal{A}$ a process $P_{\mathcal{A}}$ such that two automata yield the same language if, and only if, their corresponding processes are equivalent. This reverse encoding, from pushdown automata to protocols is explained in Section 7.4.

Our second contribution is an undecidability result. The class $\mathcal{C}_{\mathsf{pp}}$ is somewhat limited but extending $\mathcal{C}_{\mathsf{pp}}$ to non deterministic processes immediately yields undecidability of trace equivalence. More precisely, we have that trace inclusion of processes in $\mathcal{C}_{\mathsf{pp}}$ is undecidable.

Theorem 7.1.2. The following problem is undecidable.

Input P and Q two protocols in $\mathcal{C}_{\mathsf{pp}}$.

Output Whether P is trace included in Q, i.e. $P \sqsubseteq Q$.

A direct encoding of the Post Correspondence Problem (PCP) into an inclusion of two protocols of this class is given in Appendix B.1. Alternatively, this undecidability result is also a consequence of the reduction result established in Section 7.4 and the undecidability result established in [46]. Nonetheless, we present in Appendix B.1 the direct encoding of PCP into protocol equivalence since some ideas might be reused to show undecidability of trace equivalence for some other classes whereas the alternative proof required a first encoding to transform a protocol into a pushdown automaton.

Undecidability of trace inclusion actually implies undecidability of trace equivalence as soon as processes are non deterministic. Indeed consider the choice operator $+$ whose (standard) semantics is given by the following rules:

$$(\{P + Q\} \cup \mathcal{P}; \sigma) \xrightarrow{\tau} (P \cup \mathcal{P}; \sigma) \qquad (\{P + Q\} \cup \mathcal{P}; \sigma) \xrightarrow{\tau} (Q \cup \mathcal{P}; \sigma)$$

Corollary 7.1.1. Let P, Q_1, and Q_2 be three protocols in $\mathcal{C}_{\mathsf{pp}}$. The problem whether P is equivalent to $Q_1 + Q_2$, i.e. $P \approx Q_1 + Q_2$, is undecidable.

Indeed, consider P and Q_1, for which trace inclusion encodes PCP, and let $Q_2 = P$. Trivially, $P \sqsubseteq Q_1 + Q_2$. Thus $P \approx Q_1 + Q_2$ if, and only if, $Q_1 + Q_2 \sqsubseteq P$, *i.e.* if, and only if, $Q_1 \sqsubseteq P$, hence the undecidability result.

7.2 Getting rid of the full attacker

We show in this section how to reduce trace equivalence between protocols in $\mathcal{C}_{\mathsf{pp}}$ to the problem of deciding trace equivalence (still between protocols in $\mathcal{C}_{\mathsf{pp}}$) when the attacker acts as a *forwarder*, that is, when the attacker may only forward messages obtained through the protocols. This new semantics induced a new notion of trace equivalence, denoted $\approx_{\mathsf{fwd}}$, which is formally defined in Section 7.2.1.

To counterbalance the effects of this simple forwarder semantics, the key idea consists in modifying the protocols under study by adding new rules that encrypt and decrypt messages on demand for the forwarder. Formally, we define a transformation $\mathcal{T}_{\mathsf{fwd}}$ (see Section 7.2.2) that associates to a pair of protocols in $\mathcal{C}_{\mathsf{pp}}$ a finite set of pairs of protocols (still in $\mathcal{C}_{\mathsf{pp}}$), and we show the following result:

Proposition 7.2.1. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$. We have that:

$$P \approx Q \text{ if, and only if, } P' \approx_{\mathsf{fwd}} Q' \text{ for some } (P', Q') \in \mathcal{T}_{\mathsf{fwd}}(P, Q).$$

$$(\mathsf{in}(c,u).P \cup \mathcal{P};\sigma) \xrightarrow{\mathsf{in}(c,R)}_{\mathsf{fwd}} (P\theta \cup \mathcal{P};\sigma)$$
where $R \in \{\mathsf{start}\} \cup \mathcal{W}$ and $R\sigma\downarrow = u\theta$ for some θ

$$(\mathsf{out}(c,u).P \cup \mathcal{P};\sigma) \xrightarrow{\mathsf{out}(c,\mathsf{w}_{i+1})}_{\mathsf{fwd}} (P \cup \mathcal{P};\sigma \cup \{\mathsf{w}_{i+1} \triangleright u\})$$
where i is the number of elements in σ

$$(!P \cup \mathcal{P};\sigma) \xrightarrow{\tau}_{\mathsf{fwd}} (P \cup !P \cup \mathcal{P};\sigma)$$

$$(\mathsf{new}\ n.P \cup \mathcal{P};\sigma) \xrightarrow{\tau}_{\mathsf{fwd}} (P\{^{n'}/_{n}\} \cup \mathcal{P};\sigma)$$ where n' is a fresh name in $\mathcal{N}_{\mathsf{priv}}$

Figure 7.1: Semantics for a forwarder attacker.

7.2.1 Forwarder semantics

We first define the actions of a forwarder by modifying our semantics. Roughly, we restrict the recipes R, R_1, and R_2 that are used in the IN rule and in static equivalence (Definition 2.4.1) to be either the public constant start or a variable in $\mathcal{W}$. Intuitively, this corresponds to the fact that the forwarder attacker should no longer build a message on his own. This leads us to consider a new relation $\rightarrow_{\mathsf{fwd}}$ between configurations which is the relation induced by the rules described in Figure 7.1.

The relations $\xrightarrow{\mathsf{tr}}_{\mathsf{fwd}}$ and $\xRightarrow{\mathsf{tr}}_{\mathsf{fwd}}$ between configurations where tr is a sequence of actions (resp. observable actions) are defined as expected. For every configuration K, we define its *set of traces w.r.t. the forwarder semantics* as follows:

$$\mathsf{trace}_{\mathsf{fwd}}(K) = \left\{ (\mathsf{tr},\sigma) \;\middle|\; K \xRightarrow{\mathsf{tr}}_{\mathsf{fwd}} (\mathcal{P};\sigma) \text{ for some configuration } (\mathcal{P};\sigma) \text{ with } \mathsf{tr} \text{ sequence of input-output blocks.} \right\}$$

We need also to adapt our notion of static equivalence.

Definition 7.2.1. Two frames σ_1 and σ_2 are *statically equivalent w.r.t. the forwarder semantics*, denoted $\sigma_1 \sim_{\mathsf{fwd}} \sigma_2$, when we have that $dom(\sigma_1) = dom(\sigma_2)$, and for all recipes R_1 and R_2 in $\{\mathsf{start}\} \cup \mathcal{W}$, we have that $R_1\sigma_1 = R_2\sigma_1$ if, and only if, $R_1\sigma_2 = R_2\sigma_2$.

This induces a new notion of trace equivalence which is formally defined as follows:

Definition 7.2.2. Let P and Q be two protocols. We have that $P \sqsubseteq_{\mathsf{fwd}} Q$ if for every $(\mathsf{tr},\sigma) \in \mathsf{trace}_{\mathsf{fwd}}(P)$, there exists $(\mathsf{tr}',\sigma') \in \mathsf{trace}_{\mathsf{fwd}}(Q)$ such that $\mathsf{tr} = \mathsf{tr}'$ and $\sigma \sim_{\mathsf{fwd}} \sigma'$. They are *trace equivalent w.r.t. the forwarder semantics*, written $P \approx_{\mathsf{fwd}} Q$, if $P \sqsubseteq_{\mathsf{fwd}} Q$ and $Q \sqsubseteq_{\mathsf{fwd}} P$.

Example 7.2.1. The trace exhibited in Example 7.1.1 is still a valid one according to the forwarder semantics, but the frames σ_1 and σ_2 are now in equivalence w.r.t. $\sim_{\mathsf{fwd}}$. Actually, we have that $P_A \mid P_B^l \mid P_{\mathsf{key}} \approx_{\mathsf{fwd}} P_A \mid P_B^r \mid P_{\mathsf{key}}$. Indeed, the fact that a forwarder simply acts as a relay prevents him to mount the aforementioned attack.

7.2.2 Towards a forwarder attacker

As illustrated in Example 7.2.1, the forwarder semantics is very restrictive: a forwarder cannot rely on his deduction capabilities to mount an attack. We show however that we can still restrict ourselves to trace equivalence w.r.t. a forwarder.

Intuitively, we transform any two processes P, Q into processes $\bar{P}$, $\bar{Q}$ such that $P \approx Q$ if and only if $\bar{P} \approx_{\mathsf{fwd}} \bar{Q}$. Roughly this transformation consists in two steps.

1. First, we guess among the keys of the protocols P and the keys of the protocols Q those that are deducible by the attacker, as well as a bijection α between these two sets. We can show that such a bijection necessarily exists when $P \approx Q$.

2. Then, to compensate the fact that the attacker is a simple forwarder, we give him access to encryption/decryption oracles for any deducible key k, adding branches in the processes.

To maintain the equivalence, we do a similar transformation in both P and Q relying on the bijection α. We ensure that the set of deducible keys has been correctly guessed by adding of some extra processes. Then the main step of the proof consists of showing that the forwarder has now the same power as a full attacker, even though he cannot reuse the same randomness in two distinct encryptions, as a real attacker could.

Example 7.2.2. To better illustrate this section, we consider a variant of the processes introduced in Section 7.1., where agent A is now willing to talk only to B.

$$P \stackrel{\mathsf{def}}{=} P'_A \mid P^l_B \mid P_{\mathsf{key}} \qquad Q \stackrel{\mathsf{def}}{=} P'_A \mid P^r_B \mid P_{\mathsf{key}}$$

where P^l_B, P^r_B are defined in Example 7.1.1 and P_{key} is defined in Example 7.1.1, whereas P'_A is defined as follows (only the first branch of P_A)

$$P'_A \quad \stackrel{\mathsf{def}}{=} \quad !\,\mathsf{in}(c_A, \mathsf{start}).\mathsf{new}\, r^1_A.\mathsf{new}\, r^2_A.\mathsf{out}(c_A, \mathsf{raenc}(\mathsf{sign}(\mathsf{k}_{AB}, \mathsf{sk}_A, r^1_A), \mathsf{pk}_B, r^2_A))$$

This scenario excludes the aforementioned attack and we have that $P \approx Q$. This has been formally checked using our prototype (see Section 7.5).

Guessing deducible keys

The purpose of this section is to restrict our attention to protocols that explicitly disclose their deducible keys $\mathcal{K}_P$ and $\mathcal{K}_Q$. Since we do not want to rely on a particular procedure for computing these two sets, the idea is to guess a possibly superset of each set, namely K and K', and then ensure that these sets K and K' contain *at least* the deducible keys.

Definition 7.2.3. Let P be a protocol in $\mathcal{C}_{\mathsf{pp}}$. A term t is *deducible* in P if there exists a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and a recipe R (*i.e.* a term in $\mathcal{T}(\Sigma_{\mathsf{pub}}, \mathcal{N}_{\mathsf{pub}}, \mathcal{W})$) such that $R\phi\!\downarrow = t$.

Example 7.2.3. Continuing Example 7.2.2, we have that P and Q are in $\mathcal{C}_{\mathsf{pp}}$. It is easy to notice that k_{AB} is deducible in P whereas k is deducible in Q since these keys are revealed at the end of B's execution. For both P and Q, the trace $\mathsf{tr} = \mathsf{in}(c_A, \mathsf{start}).\mathsf{out}(c_A, \mathsf{w}_1).\mathsf{in}(c_B, \mathsf{w}_1).\mathsf{out}(c_B, w_2)$ and the recipe $R = \mathsf{w}_2$ is a witness of this fact.

Two equivalent processes have the same set of deducible keys, up to some bijective renaming.

Lemma 7.2.1. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$, $\mathcal{K}_P$ (resp. $\mathcal{K}_Q$) be the set of deducible constants of sort key that occur in P (resp. Q), if $P \approx Q$ then there exists a unique bijection α from $\mathcal{K}_P$ to $\mathcal{K}_Q$ such that for every trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ there exists a trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ such that for any recipe R and any $\mathsf{k} \in \mathcal{K}_P$:

- $R\phi\!\downarrow$ is of sort s if, and only if, $R\psi\!\downarrow$ is of sort s;

  where $\mathsf{s} \in \{\mathsf{SymKey}, \mathsf{PubKey}, \mathsf{PrivKey}\}$.
- $R\phi\!\downarrow = \mathsf{k}$ if, and only if, $R\psi\!\downarrow = \alpha(\mathsf{k})$;
- $R\phi\!\downarrow = \mathsf{k}^{-1}$ if, and only if, $R\psi\!\downarrow = (\alpha(\mathsf{k}))^{-1}$;

and conversely, for every $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ there exists a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ satisfying the same properties.

Proof. (sketch) The relation α is defined as follows:

> for every $\mathsf{k} \in \mathcal{K}_P$ of sort s, and every trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and recipe R such that $R\phi\!\downarrow = \mathsf{k}$, we define $\alpha(\mathsf{k}) = R\psi\!\downarrow$ where ψ is the only frame such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$.

The existence of such a frame comes from the fact that $P \approx Q$, whereas its unicity is a consequence of the determinism of protocols in $\mathcal{C}_{\mathsf{pp}}$.

Then, we show that this relation α is uniquely defined and satisfied all the requirements exploiting the strong relationship between P and Q through the relation $P \approx Q$. □

Example 7.2.4. Continuing Example 7.2.2, we have $\mathcal{K}_P = \{\mathsf{pk}_A, \mathsf{pk}_B, \mathsf{pk}_C, \mathsf{sk}_C, \mathsf{k}_{AB}\}$ whereas $\mathcal{K}_Q = \{\mathsf{pk}_A, \mathsf{pk}_B, \mathsf{pk}_C, \mathsf{sk}_C, \mathsf{k}\}$. The unique bijection α mentioned in the previous lemma is defined as follows: $\alpha(\mathsf{k}_{AB}) = \mathsf{k}$, and $\alpha(\mathsf{k}') = \mathsf{k}'$ otherwise.

Definition 7.2.4. Let P be a protocol in $\mathcal{C}_{\mathsf{pp}}$, K be a set of constants of sort key that occur in P. If for every $\mathsf{k} \in K$ there exist a channel name c_{k} and a branch $!\mathsf{in}(c_{\mathsf{k}}, \mathsf{start}).\mathsf{out}(c_{\mathsf{k}}, \mathsf{k})$ in P, then P is said *to disclose* K.

Example 7.2.5. Continuing our running example, P and Q clearly disclose $K = \{\mathsf{pk}_A, \mathsf{pk}_B, \mathsf{pk}_C, \mathsf{sk}_C\}$.

Lemma 7.2.2. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$, S (resp. S') the set of keys of P (resp. Q). Then $P \approx Q$ if, and only if, there exist two sets $K \subseteq S$ and $K' \subseteq S'$ and a bijection $\alpha : K \to K'$ such that $\bar{P} \approx \bar{Q}$ where:

$$\begin{aligned}
\bar{P} = P \quad &| \quad !\mathsf{in}(c^0, \mathsf{start}).\mathsf{out}(c^0, 0) | !\mathsf{in}(c^1, \mathsf{start}).\mathsf{out}(c^1, 1) \\
&| \quad \Big|_{\mathsf{k} \in K} !\mathsf{in}(c_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{start}).\mathsf{out}(c_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{k}) | \Big|_{\mathsf{k} \in S \setminus K} !\mathsf{in}(c, \mathsf{k}).\mathsf{out}(c, 0) \\
\bar{Q} = Q \quad &| \quad !\mathsf{in}(c^0, \mathsf{start}).\mathsf{out}(c^0, 0) | !\mathsf{in}(c^1, \mathsf{start}).\mathsf{out}(c^1, 1) \\
&| \quad \Big|_{\mathsf{k} \in K} !\mathsf{in}(c_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{start}).\mathsf{out}(c_{\mathsf{k},\alpha(\mathsf{k})}, \alpha(\mathsf{k})) | \Big|_{\mathsf{k} \in S' \setminus K'} !\mathsf{in}(c, \mathsf{k}).\mathsf{out}(c, 1)
\end{aligned}$$

and 0, 1 are new constants, c^0, c^1, the $c_{\mathsf{k},\alpha(\mathsf{k})}$ and c are fresh channels.

Moreover, assuming the existence of such sets and bijection such that $\bar{P} \approx \bar{Q}$, the two protocols are disclosing their deducible keys.

We call $\mathcal{T}_{\mathsf{key}}(P, Q)$ the set of such pairs $(\bar{P}, \bar{Q})$ of modified protocols.

Proof. Let $\mathcal{K}_P$ (resp. $\mathcal{K}_Q$) be the set of deducible constants of sort key that occur in P (resp. Q). We prove the two directions separately.

($\Rightarrow$) If $P \approx Q$, by Lemma 7.2.1, for $K = \mathcal{K}_P$ and $K' = \mathcal{K}_Q$, we get the existence of such a bijection α. Because keys in $S \setminus \mathcal{K}_P$ and $S' \setminus \mathcal{K}_Q$ are not deducible, the branches on channel c can never be triggered. Moreover, as $P \approx Q$, any trace of P (resp. Q) inputting or outputting on a channel $c_{\mathsf{k},\alpha(\mathsf{k})}$ for k in $\mathcal{K}_P$ can be matched in Q (resp. P). Indeed, for every couple $(\mathsf{k}, \mathsf{k}^{-1})$ of deducible keys and for any recipe reducing to k (resp. k^{-1}) in P, the same recipe reduces to $\alpha(\mathsf{k})$ (resp. $\alpha(\mathsf{k})^{-1}$) in Q, thanks to the properties of α described in Lemma 7.2.1.

($\Leftarrow$) For the converse implication, we first remark that necessarily we have that $\mathcal{K}_P \subseteq K$ and $\mathcal{K}_Q \subseteq K'$. Indeed, suppose there exists, for instance, $\mathsf{k} \in \mathcal{K}_P \setminus K$. Since k is deducible, there exists a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and a recipe R such that $R\phi\downarrow = \mathsf{k}$. Since (tr, ϕ) is also a trace of $\bar{P}$, we consider the trace:

$$\mathsf{tr}' = \mathsf{tr}.\mathsf{in}(c, R).\mathsf{out}(c, \mathsf{w}_{|\phi|+1}).\mathsf{in}(c^0, \mathsf{start}).\mathsf{out}(c^0, \mathsf{w}_{|\phi|+2}).\mathsf{in}(c^1, \mathsf{start}).\mathsf{out}(c^1, \mathsf{w}_{|\phi|+3})$$

along with its frame $\phi' = \phi \cup \{\mathsf{w}_{|\phi|+1} \triangleright 0, \mathsf{w}_{|\phi|+2} \triangleright 0, \mathsf{w}_{|\phi|+3} \triangleright 1\}$. If $\bar{P} \approx \bar{Q}$, then there exists $(\mathsf{tr}', \psi') \in \mathsf{trace}(\bar{Q})$ such that ϕ and ψ are statically equivalent. But any output on c in Q leads to the constant 1, breaking static equivalence. We conclude in a similar way in case $\mathsf{k} \in \mathcal{K}_Q \setminus K'$.

Finally we need to prove that $\bar{P} \approx \bar{Q}$ implies $P \approx Q$. For every trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, $(\mathsf{tr}, \phi) \in \mathsf{trace}(\bar{P})$, and as $\bar{P} \approx \bar{Q}$, there exists a trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(\bar{Q})$ such that ϕ is statically equivalent to ψ. Because c^0, c^1, c and the $c_{\mathsf{k},\alpha(\mathsf{k})}$ are new channels, tr does not use transitions on those, thus $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$. The same goes for any trace of Q, hence showing the trace equivalence of P and Q. □

Example 7.2.6. Continuing our example, let $K = \mathcal{K}_P$ and $K' = \mathcal{K}_Q$, and α the bijection defined in Example 7.2.4. Checking equivalence of $P \approx Q$ amounts into checking whether $\bar{P} \approx \bar{Q}$ where $\bar{P}$ and $\bar{Q}$ are defined as follows.

$$\begin{aligned}
\bar{P} = P \quad &| \quad !\mathsf{in}(c^0, \mathsf{start}).\mathsf{out}(c^0, 0) | !\mathsf{in}(c^1, \mathsf{start}).\mathsf{out}(c^1, 1) \\
&| \quad !\mathsf{in}(c_{\mathsf{k}_{AB},\mathsf{k}}, \mathsf{start}).\mathsf{out}(c_{\mathsf{k}_{AB},\mathsf{k}}, \mathsf{k}_{AB}) \\
&| \quad !\mathsf{in}(c, \mathsf{sk}_A).\mathsf{out}(c, 0) \ | !\mathsf{in}(c, \mathsf{sk}_B).\mathsf{out}(c, 0) \\
\\
\bar{Q} = Q \quad &| \quad !\mathsf{in}(c^0, \mathsf{start}).\mathsf{out}(c^0, 0) | !\mathsf{in}(c^1, \mathsf{start}).\mathsf{out}(c^1, 1) \\
&| \quad !\mathsf{in}(c_{\mathsf{k}_{AB},\mathsf{k}}, \mathsf{start}).\mathsf{out}(c_{\mathsf{k}_{AB},\mathsf{k}}, \mathsf{k}) \\
&| \quad !\mathsf{in}(c, \mathsf{sk}_A).\mathsf{out}(c, 1) \ | !\mathsf{in}(c, \mathsf{sk}_B).\mathsf{out}(c, 1)
\end{aligned}$$

If $\bar{P} \approx \bar{Q}$, then sk_A and sk_B cannot be deducible thus $\bar{P}$ and $\bar{Q}$ disclose their set of deducible keys.

Adding oracles

To compensate the fact that the attacker is a simple forwarder, we give him access to encryption/decryption oracles for any deducible key k, adding branches in the processes. We rely on the bijection α computed in the previous section to do this in a compatible way on both sides of the equivalence.

Lemma 7.2.3. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$ respectively disclosing two sets of keys K and K' as in Lemma 7.2.2. Then $P \approx Q$ if, and only if, $\bar{P} \approx_{\mathsf{fwd}} \bar{Q}$ where:

$$\begin{aligned}
\bar{P} = P \quad &| \quad \Big|_{\mathsf{k} \in K^{\mathsf{SymKey}}} !\mathsf{in}(c^{\mathsf{senc}}_{\mathsf{k},\alpha(\mathsf{k})}, x).\mathsf{new}\ n.\mathsf{out}(c_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{senc}(x, \mathsf{k}, n)) \\
&| \quad \Big|_{\mathsf{k} \in K^{\mathsf{SymKey}}} !\mathsf{in}(c^{\mathsf{sdec}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{senc}(x, \mathsf{k}, y)).\mathsf{out}(c^{\mathsf{sdec}}_{\mathsf{k},\alpha(\mathsf{k})}, x) \\
&| \quad \Big|_{\mathsf{k} \in K^{\mathsf{PubKey}}} !\mathsf{in}(c^{\mathsf{raenc}}_{\mathsf{k},\alpha(\mathsf{k})}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{raenc}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{raenc}(x, \mathsf{k}, n)) \\
&| \quad \Big|_{\mathsf{k} \in K^{\mathsf{PrivKey}}} !\mathsf{in}(c^{\mathsf{radec}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{raenc}(x, \mathsf{k}, y)).\mathsf{out}(c^{\mathsf{radec}}_{\mathsf{k},\alpha(\mathsf{k})}, x) \\
&| \quad \Big|_{\mathsf{k} \in K^{\mathsf{PrivKey}}} !\mathsf{in}(c^{\mathsf{sign}}_{\mathsf{k},\alpha(\mathsf{k})}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{sign}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{sign}(x, \mathsf{k}, n)) \\
&| \quad \Big|_{\mathsf{k} \in K^{\mathsf{PubKey}}} !\mathsf{in}(c^{\mathsf{check}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{sign}(x, \mathsf{k}, y)).\mathsf{out}(c^{\mathsf{check}}_{\mathsf{k},\alpha(\mathsf{k})}, x) \\
\\
\bar{Q} = Q \quad &| \quad \Big|_{\mathsf{k} \in K^{\mathsf{SymKey}}} !\mathsf{in}(c^{\mathsf{senc}}_{\mathsf{k},\alpha(\mathsf{k})}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{senc}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{senc}(x, \alpha(\mathsf{k}), n)) \\
&| \quad \Big|_{\mathsf{k} \in K^{\mathsf{SymKey}}} !\mathsf{in}(c^{\mathsf{sdec}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{senc}(x, \alpha(\mathsf{k}), y)).\mathsf{out}(c^{\mathsf{sdec}}_{\mathsf{k},\alpha(\mathsf{k})}, x) \\
&| \quad \Big|_{\mathsf{k} \in K^{\mathsf{PubKey}}} !\mathsf{in}(c^{\mathsf{raenc}}_{\mathsf{k},\alpha(\mathsf{k})}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{raenc}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{raenc}(x, \alpha(\mathsf{k}), n)) \\
&| \quad \Big|_{\mathsf{k} \in K^{\mathsf{PrivKey}}} !\mathsf{in}(c^{\mathsf{radec}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{raenc}(x, \alpha(\mathsf{k}), y)).\mathsf{out}(c^{\mathsf{radec}}_{\mathsf{k},\alpha(\mathsf{k})}, x) \\
&| \quad \Big|_{\mathsf{k} \in K^{\mathsf{PrivKey}}} !\mathsf{in}(c^{\mathsf{sign}}_{\mathsf{k},\alpha(\mathsf{k})}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{sign}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{sign}(x, \alpha(\mathsf{k}), n)) \\
&| \quad \Big|_{\mathsf{k} \in K^{\mathsf{PubKey}}} !\mathsf{in}(c^{\mathsf{check}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{check}(x, \alpha(\mathsf{k}), y)).\mathsf{out}(c^{\mathsf{check}}_{\mathsf{k},\alpha(\mathsf{k})}, x)
\end{aligned}$$

where K^{s} denotes the keys of sort s of K. We call $\mathcal{T}_{\mathsf{oracle}}$ the transformation taking a pair of protocols (P, Q) satisfying the aforementioned condition and returning the pair $(\bar{P}, \bar{Q})$ presently defined.

Proof. (sketch) First, thanks to Lemma 7.2.2, we know that P, $\bar{P}$, Q and $\bar{Q}$ disclose all their deducible keys.

($\Rightarrow$) Given a witness of non-equivalence for $\bar{P} \approx_{\mathsf{fwd}} \bar{Q}$, it is quite easy to build a witness of non-equivalence for $P \not\approx Q$ replacing the use of the oracle by the corresponding attacker construction. This yields a witness of non-equivalence for $P \approx Q$.

($\Leftarrow$) This direction is actually more involved. The idea is to replace the use of an attacker construction, e.g. an encryption with a deducible key, by the corresponding oracle. However, the attacker has the ability to use the same random seed more than once whereas this is impossible when using the oracles to perform those computations. Thus, we first show that this additional ability does not give any power to the attacker. Then, we do the replacement as expected in order to conclude.

The full proof is provided in Appendix B.2.

□

Example 7.2.7. Continuing our example, this last transformation will add 10 branches (2 per deducible key). For instance, regarding the key k_{AB}, the two following branches will be be added:

For process P:

$$\begin{aligned} & !\mathsf{in}(c^{\mathsf{senc}}_{\mathsf{k}_{AB},\mathsf{k}}, x).\mathsf{new}\ n.\mathsf{out}(c_{\mathsf{k}_{AB},\mathsf{k}}, \mathsf{senc}(x, \mathsf{k}_{AB}, n)) \\ \mid\ & !\mathsf{in}(c^{\mathsf{sdec}}_{\mathsf{k}_{AB},\mathsf{k}}, \mathsf{senc}(x, \mathsf{k}_{AB}, y)).\mathsf{out}(c^{\mathsf{sdec}}_{\mathsf{k}_{AB},\mathsf{k}}, x) \end{aligned}$$

For process Q:

$$\begin{aligned} & !\mathsf{in}(c^{\mathsf{senc}}_{\mathsf{k}_{AB},\mathsf{k}}, x).\mathsf{new}\ n.\mathsf{out}(c_{\mathsf{k}_{AB},\mathsf{k}}, \mathsf{senc}(x, \mathsf{k}, n)) \\ \mid\ & !\mathsf{in}(c^{\mathsf{sdec}}_{\mathsf{k}_{AB},\mathsf{k}}, \mathsf{senc}(x, \mathsf{k}, y)).\mathsf{out}(c^{\mathsf{sdec}}_{\mathsf{k}_{AB},\mathsf{k}}, x) \end{aligned}$$

Regarding the keys $\mathsf{pk}_A, \mathsf{pk}_B, \mathsf{pk}_C$ and sk_C, since $\alpha(\mathsf{k}') = \mathsf{k}'$ for each of these keys, we add the following branches on both sides:

$$\begin{aligned} \mid\ & !\mathsf{in}(c^{\mathsf{raenc}}_{\mathsf{pk}_A,pk_A}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{raenc}}_{\mathsf{pk}_A,\mathsf{pk}_A}, \mathsf{raenc}(x, \mathsf{pk}_A, n)) \\ \mid\ & !\mathsf{in}(c^{\mathsf{raenc}}_{\mathsf{pk}_B,pk_B}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{raenc}}_{\mathsf{pk}_B,\mathsf{pk}_B}, \mathsf{raenc}(x, \mathsf{pk}_B, n)) \\ \mid\ & !\mathsf{in}(c^{\mathsf{raenc}}_{\mathsf{pk}_C,pk_C}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{raenc}}_{\mathsf{pk}_C,\mathsf{pk}_C}, \mathsf{raenc}(x, \mathsf{pk}_C, n)) \\ \mid\ & !\mathsf{in}(c^{\mathsf{radec}}_{\mathsf{sk}_C,\mathsf{sk}_C}, \mathsf{raenc}(x, \mathsf{pk}_C, y)).\mathsf{out}(c^{\mathsf{radec}}_{\mathsf{sk}_C,\mathsf{sk}_C}, x) \\ \mid\ & !\mathsf{in}(c^{\mathsf{sign}}_{\mathsf{sk}_C,\mathsf{sk}_C}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{sign}}_{\mathsf{sk}_C,\mathsf{sk}_C}, \mathsf{sign}(x, \mathsf{sk}_C, n)) \\ \mid\ & !\mathsf{in}(c^{\mathsf{check}}_{\mathsf{pk}_A,\mathsf{pk}_A}, \mathsf{sign}(x, \mathsf{sk}_A, y)).\mathsf{out}(c^{\mathsf{check}}_{\mathsf{pk}_A,\mathsf{pk}_A}, x) \\ \mid\ & !\mathsf{in}(c^{\mathsf{check}}_{\mathsf{pk}_B,\mathsf{pk}_B}, \mathsf{sign}(x, \mathsf{sk}_B, y)).\mathsf{out}(c^{\mathsf{check}}_{\mathsf{pk}_B,\mathsf{pk}_B}, x) \\ \mid\ & !\mathsf{in}(c^{\mathsf{check}}_{\mathsf{pk}_C,\mathsf{pk}_C}, \mathsf{sign}(x, \mathsf{sk}_C, y)).\mathsf{out}(c^{\mathsf{check}}_{\mathsf{pk}_C,\mathsf{pk}_C}, x) \end{aligned}$$

Transformation $\mathcal{T}_{\mathsf{fwd}}$

Thanks to Lemmas 7.2.2 and 7.2.3, we are now able to formally define our transformation that gets rid of a fully active attacker. For every pair of protocols (P, Q) in $\mathcal{C}_{\mathsf{pp}}$, we consider

$$\mathcal{T}_{\mathsf{fwd}}(P, Q) = \{\mathcal{T}_{\mathsf{oracle}}(P', Q') \mid (P', Q') \in \mathcal{T}_{\mathsf{key}}(P, Q)\}$$

Combination of the two previous results yields to the desired result.

Proposition 7.2.1. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$. We have that:

$$P \approx Q \text{ if, and only if, } P' \approx_{\mathsf{fwd}} Q' \text{ for some } (P', Q') \in \mathcal{T}_{\mathsf{fwd}}(P, Q).$$

7.3 Encoding protocols into real-time GPDAs

We first introduce the notion of real-time generalised pushdown automaton (GPDA) (see Section 7.3.1) before explaining in details (see Sections 7.3.2 and 7.3.3) our encoding from protocols to real-time generalised pushdown automata.

More precisely, for any process $P \in \mathcal{C}_{\mathsf{pp}}$, we show that it is possible to define a polynomial-sized real-time generalised pushdown automaton $\mathcal{A}_P$ such that trace equivalence w.r.t. the forwarder semantics coincides with language equivalence of the two corresponding automata.

Theorem 7.3.1. Let P and Q in $\mathcal{C}_{\mathsf{pp}}$, we have that:

$$P \approx_{\mathsf{fwd}} Q \iff \mathcal{L}(\mathcal{A}_P) = \mathcal{L}(\mathcal{A}_Q).$$

The proof of this theorem consists of three main steps.

1. First, we provide a new characterisation of trace equivalence w.r.t. the forwarder semantics. Intuitively, we show that it is not necessary to consider all possible tests (when checking static equivalence). Indeed, our Lemma 7.3.1 states that it is sufficient to check for constant tests (that is, tests of the form $x = c$ where c is a constant) and some specific class of tests that we call *guarded* and *pulled-up*.

2. Then we associate to processes $P, Q \in \mathcal{C}_{\mathsf{pp}}$ real-time GPDAs that check whether they satisfy the same constant tests (Lemma 7.3.2).

3. And we associate to processes $P, Q \in \mathcal{C}_{\mathsf{pp}}$ real-time GPDAs that check whether they satisfy the same guarded tests (Lemma 7.3.3).

All along this section, we illustrate the definitions with the protocol displayed in Figure 7.2. This example should be read step by step, when reading the examples of this section.

7.3.1 Generalised pushdown automata

Language equivalence of deterministic pushdown automata (DPA) is known to be decidable [63]. We actually encode equivalence of protocols into a fragment of DPA: real-time GPDA with final-state acceptance. GPDA differ from deterministic pushdown automata (DPA) as they can unstack several symbols at a time. Real-time automata are automata that do not include epsilon-transitions. Formally, the class of real-time GPDA is defined as follows.

Definition 7.3.1. A *real-time GPDA* is a 7-tuple $\mathcal{A} = (Q, \Pi, \Gamma, q_0, \omega, Q_f, \delta)$ where Q is a finite set of states, $q_0 \in Q$ is an initial state, $Q_f \subseteq Q$ is a set of accepting states, Π is a finite input-alphabet, Γ is a finite stack-alphabet, ω is the initial stack symbol, and $\delta : (Q \times \Pi \times \Gamma_0) \to Q \times \Gamma_0$ is a partial transition function such that:

- Γ_0 is a finite subset of Γ^*; and
- for any $(q, a, x) \in dom(\delta)$ and y suffix strict of x, we have that $(q, a, y) \notin dom(\delta)$.

Let $q, q' \in Q$, $u, u', \gamma \in \Gamma^*$, $m \in \Pi^*$, $a \in \Pi$; we note $(qu\gamma, am) \rightsquigarrow_{\mathcal{A}} (q'uu', m)$ if $(q', u') = \delta(q, a, \gamma)$. The relation $\rightsquigarrow^*_{\mathcal{A}}$ is the reflexive and transitive closure of $\rightsquigarrow_{\mathcal{A}}$. For every qu, $q'u'$ in $Q\Gamma^*$ and $m \in \Pi^*$, we note $qu \xrightarrow{m}_{\mathcal{A}} q'u'$ if, and only if, $(qu, m) \rightsquigarrow^*_{\mathcal{A}} (q'u', \epsilon)$. For sake of clarity, a transition from q to q' reading a, popping γ from the stack and pushing u' will be denoted by $q \xrightarrow{a;\gamma/u'} q'$.

Let $\mathcal{A}$ be a GPDA. The language recognised by $\mathcal{A}$ is defined by:

$$\mathcal{L}(\mathcal{A}) = \{m \in \Pi^* \mid q_0\omega\mathsf{start} \xrightarrow{m}_{\mathcal{A}} q_f u \text{ for some } q_f \in Q_f \text{ and } u \in \Gamma^*\}.$$

Note that the language is defined starting with the word $\omega\mathsf{start}$ in the stack.

A real-time GPDA can easily be converted into a DPA by adding new states and ϵ-transitions. Thus, the problem of language equivalence for two real-time GPDA $\mathcal{A}_1$ and $\mathcal{A}_2$, *i.e.* deciding whether $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$ is decidable [63]. Whether deciding equivalence of real-time GPDA could be easier than deciding equivalence of DPA is an open question.

We illustrate the different steps of our translation of protocols to automata using a (mock) ping-pong protocol P_{toy}. We define $\mathsf{io}(c, R, \mathsf{w}) \stackrel{\mathsf{def}}{=} \mathsf{in}(c, R).\mathsf{out}(c, \mathsf{w})$.

$$
\begin{aligned}
&P_{\mathsf{toy}} = \\
&\quad | \,!\, \mathsf{in}(c_1, \mathsf{start}).\mathsf{new}\ r_1.\mathsf{out}(c_1, \mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_1)) \\
&\quad | \,!\, \mathsf{in}(c_2, \mathsf{senc}(x, \mathsf{k}_2, z_1)).\mathsf{new}\ r_1.\mathsf{out}(c_2, \mathsf{senc}(x, \mathsf{k}_1, r_1)) \\
&\quad | \,!\, \mathsf{in}(c_3, x).\mathsf{new}\ r_1.\mathsf{out}(c_3, \mathsf{senc}(x, \mathsf{k}_2, r_1)) \\
&\quad | \,!\, \mathsf{in}(c_4, \mathsf{senc}(\mathsf{senc}(x, \mathsf{k}_1, z_1), \mathsf{k}_2, z_2)).\mathsf{new}\ r_1, r_2.\mathsf{out}(c_4, \mathsf{senc}(\mathsf{senc}(x, \mathsf{k}_2, r_1), \mathsf{k}_1, r_2)) \\
&\quad | \,!\, \mathsf{in}(c_5, \mathsf{senc}(\mathsf{senc}(x, \mathsf{k}_2, z_1), \mathsf{k}_1, z_2)).\mathsf{out}(c_5, x)
\end{aligned}
$$

For illustrative purpose, we consider different execution traces of this protocol. For instance, we have that $(\mathsf{tr}_1, \sigma_1) \in \mathsf{trace}_{\mathsf{fwd}}(P_{\mathsf{toy}})$ where:

- $\mathsf{tr}_1 = \mathsf{io}(c_1, \mathsf{start}, \mathsf{w}_1).\mathsf{io}(c_2, \mathsf{w}_1, \mathsf{w}_2).\mathsf{io}(c_3, \mathsf{w}_2, \mathsf{w}_3).\mathsf{io}(c_4, \mathsf{w}_3, \mathsf{w}_4).\mathsf{io}(c_5, \mathsf{w}_4, \mathsf{w}_5)$, and
- $\sigma_1 = \{\mathsf{w}_1 \triangleright \mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_1),\ \mathsf{w}_2 \triangleright \mathsf{senc}(\mathsf{a}, \mathsf{k}_1, r_2),\ \mathsf{w}_3 \triangleright \mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_1, r_2), \mathsf{k}_2, r_3),$
  $\mathsf{w}_4 \triangleright \mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4), \mathsf{k}_1, r_5),\ \mathsf{w}_5 \triangleright \mathsf{a}\}$.

This execution may be continued as follows:

- $\mathsf{tr}_2 = \mathsf{io}(c_3, \mathsf{w}_1, \mathsf{w}_6).\mathsf{io}(c_2, \mathsf{w}_6, \mathsf{w}_7).\mathsf{io}(c_5, \mathsf{w}_7, \mathsf{w}_8)$, and
- $\sigma_2 = \{\mathsf{w}_6 \triangleright \mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_1), \mathsf{k}_2, r_6),\ \mathsf{w}_7 \triangleright \mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_1), \mathsf{k}_1, r_7),$
  $\mathsf{w}_8 \triangleright \mathsf{a}\}$.

Let $\sigma_{1/2} = \sigma_1 \cup \sigma_2$. We have that $(\mathsf{tr}_1.\mathsf{tr}_2, \sigma_{1/2})$ is a trace of P_{toy} w.r.t. the forwarder semantics. We have that the test $\mathsf{w}_5 = \mathsf{w}_8$ is valid in $\sigma_{1/2}$. Indeed $\mathsf{w}_5\sigma_{1/2}{\downarrow} = \mathsf{w}_8\sigma_{1/2}{\downarrow} = \mathsf{a}$.

We have also that $(\mathsf{tr}'_1, \sigma'_1) \in \mathsf{trace}_{\mathsf{fwd}}(P_{\mathsf{toy}})$ with:

- $\mathsf{tr}'_1 = \mathsf{io}(c_1, \mathsf{start}, \mathsf{w}_1).\mathsf{io}(c_2, \mathsf{w}_1, \mathsf{w}_2).\mathsf{io}(c_3, \mathsf{w}_2, \mathsf{w}_3).\mathsf{io}(c_4, \mathsf{w}_3, \mathsf{w}_4).\mathsf{io}(c_3, \mathsf{w}_4, \mathsf{w}_5)$, and
- $\sigma'_1 = \{\mathsf{w}_1 \triangleright \mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_1),\ \mathsf{w}_2 \triangleright \mathsf{senc}(\mathsf{a}, \mathsf{k}_1, r_2),$
  $\mathsf{w}_3 \triangleright \mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_1, r_2), \mathsf{k}_2, r_3),\ \mathsf{w}_4 \triangleright \mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4), \mathsf{k}_1, r_5),$
  $\mathsf{w}_5 \triangleright \mathsf{senc}(\mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4), \mathsf{k}_1, r_5), \mathsf{k}_2, r_6)\}$.

This execution may be continued as follows:

- $\mathsf{tr}'_2 = \mathsf{io}(c_4, \mathsf{w}_5, \mathsf{w}_6).\mathsf{io}(c_5, \mathsf{w}_6, \mathsf{w}_7).\mathsf{io}(c_4, \mathsf{w}_5, \mathsf{w}_8).\mathsf{io}(c_5, \mathsf{w}_8.\mathsf{w}_9)$,
- $\sigma'_2 = \{\mathsf{w}_6 \triangleright \mathsf{senc}(\mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4), \mathsf{k}_2, r_7), \mathsf{k}_1, r_8),\ \mathsf{w}_7 \triangleright \mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4),$
  $\mathsf{w}_8 \triangleright \mathsf{senc}(\mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4), \mathsf{k}_2, r'_7), \mathsf{k}_1, r'_8),\ \mathsf{w}_9 \triangleright \mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4)\}$.

Let $\sigma'_{1/2} = \sigma'_1 \cup \sigma'_2$. We have that $(\mathsf{tr}'_1.\mathsf{tr}'_2, \sigma'_{1/2})$ is a trace of P_{toy} w.r.t. the forwarder semantics. We have that the test $\mathsf{w}_7 = \mathsf{w}_9$ is valid in $\sigma'_{1/2}$.

Figure 7.2: Running example.

7.3.2 Characterisation of trace equivalence

To construct the automaton associated to a process $P \in \mathcal{C}_{\mathsf{pp}}$, we need to construct an automaton that recognises any execution of P and the corresponding valid tests.

We first propose a new characterisation of trace equivalence allowing us to restrict our attention to executions of P and valid tests that have a special form.

Given a trace (tr, σ) and an element w of a frame σ, we can extract from tr the sequence of actions that conducted to the production of this element w.

Definition 7.3.2. Let P be a protocol in $\mathcal{C}_{\mathsf{pp}}$, tr be a trace of P w.r.t. the forwarder semantics, *i.e.* such that $(\mathsf{tr}, \sigma) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ for some σ, and w be a variable that occurs in tr. The *sequence associated to* w *in* tr, denoted $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$, is the subsequence of tr of the following form:

$$\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}) = \mathsf{io}(c_{i_0}, \mathsf{start}, \mathsf{w}_{j_0}).\mathsf{io}(c_{i_1}, \mathsf{w}_{j_0}, \mathsf{w}_{j_1}) \ldots \mathsf{io}(c_{i_p}, \mathsf{w}_{j_{p-1}}, \mathsf{w}).$$

Example 7.3.1. Consider the protocol defined in Figure 7.2. Then,

- $\mathsf{seq}_{\mathsf{tr}_1.\mathsf{tr}_2}(\mathsf{w}_5) = \mathsf{tr}_1$;
- $\mathsf{seq}_{\mathsf{tr}_1.\mathsf{tr}_2}(\mathsf{w}_8) = \mathsf{io}(c_1, \mathsf{start}, \mathsf{w}_1).\mathsf{tr}_2$;
- $\mathsf{seq}_{\mathsf{tr}'_1.\mathsf{tr}'_2}(\mathsf{w}_7) = \mathsf{tr}'_1.\mathsf{io}(c_4, \mathsf{w}_5.\mathsf{w}_6).\mathsf{io}(c_5, \mathsf{w}_6, \mathsf{w}_7)$;
- $\mathsf{seq}_{\mathsf{tr}'_1.\mathsf{tr}'_2}(\mathsf{w}_9) = \mathsf{tr}'_1.\mathsf{io}(c_4, \mathsf{w}_5, \mathsf{w}_8).\mathsf{io}(c_5, \mathsf{w}_8, \mathsf{w}_9)$.

We consider some particular class of tests, called *pulled-up* tests.

Definition 7.3.3. Let P be a protocol in $\mathcal{C}_{\mathsf{pp}}$, $(\mathsf{tr}, \sigma) \in \mathsf{trace}_{\mathsf{fwd}}(P)$, and $\mathsf{w}, \mathsf{w}' \in dom(\sigma)$ such that:

1. the test $\mathsf{w} = \mathsf{w}'$ is σ-valid, *i.e.* $\mathsf{w}\sigma = \mathsf{w}'\sigma$; and
2. the test $\mathsf{w} = \mathsf{w}'$ is σ-guarded, *i.e.* the head symbol of $\mathsf{w}\sigma$ (or equivalently $\mathsf{w}'\sigma$) is in $\{\mathsf{senc}, \mathsf{raenc}, \mathsf{sign}\}$.

Let $\mathsf{io}(c_{i_0}, \mathsf{start}, \mathsf{w}_{j_0}) \ldots \mathsf{io}(c_{i_p}, \mathsf{w}_{j_{p-1}}, \mathsf{w}_{j_p})$ be the maximal common prefix of $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$. The test $\mathsf{w} = \mathsf{w}'$ is said to be *pulled-up* in (tr, σ) if $p = 0$, or $p \geq 1$ and $\mathsf{w}\sigma$ does not occur as a subterm in $\mathsf{w}_{j_0}\sigma, \ldots, \mathsf{w}_{j_{p-1}}\sigma$.

Intuitively, to perform a test $\mathsf{w} = \mathsf{w}'$, the attacker (who acts as a forwarder) relies on the protocol rules to produce successive outputs, and ultimately the ones stored in w and w'. The attacker may produce w and w' independently (the common prefix of $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$ is empty), and in such a case the test is pulled-up by definition. This is not, of course, always possible. In particular, a test $\mathsf{w} = \mathsf{w}'$ satisfying conditions (1) and (2) of the previous definition is necessarily a "forked" test, *i.e.* a test for which the common prefix of $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$ is not reduced to the empty sequence, and thus $p \geq 1$. Indeed, $\mathsf{w}\sigma$ is a term of the form $\mathsf{f}(u, \mathsf{k}, r)$ with some random r. Since nonces are uniquely generated, the variables w_i that generates it, *i.e.* the smallest i such that r occurs in $\mathsf{w}_i\sigma$, occurs both in $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$. For this kind of "forked" test, we can restrict the attacker to consider tests that are pulled-up, i.e. we consider tests for which the size of the common prefix between $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$ is reduced to the minimum. This can be done by duplicating some execution steps since all the branches are under a replication.

Example 7.3.2. Continuing our running example, we have that $\mathsf{w}_5 = \mathsf{w}_8$ is a test that is $\sigma_{1/2}$-valid but it is not $\sigma_{1/2}$-guarded since $\mathsf{w}_5\sigma_{1/2} = \mathsf{w}_8\sigma_{1/2} = \mathsf{a}$.

The test $\mathsf{w}_7 = \mathsf{w}_9$ is a test that is $\sigma'_{1/2}$-valid and $\sigma'_{1/2}$-guarded. Indeed, we have that $\mathsf{w}_7\sigma'_{1/2} = \mathsf{w}_9\sigma'_{1/2} = \mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4)$. The maximal common prefix of $\mathsf{seq}_{\mathsf{tr}'_1.\mathsf{tr}'_2}(\mathsf{w}_7)$ and $\mathsf{seq}_{\mathsf{tr}'_1.\mathsf{tr}'_2}(\mathsf{w}_9)$ is actually

$$\mathsf{tr}'_1 = \mathsf{io}(c_1, \mathsf{start}, \mathsf{w}_1).\mathsf{io}(c_2, \mathsf{w}_1, \mathsf{w}_2).\mathsf{io}(c_3, \mathsf{w}_2, \mathsf{w}_3).\mathsf{io}(c_4, \mathsf{w}_3, \mathsf{w}_4).\mathsf{io}(c_3, \mathsf{w}_4, \mathsf{w}_5).$$

Actually, $\mathsf{w}_7\sigma'_{1/2}$ occurs as a subterm in $\mathsf{w}_4\sigma'_{1/2}$, thus the test $\mathsf{w}_7 = \mathsf{w}_9$ is not pulled-up in $(\mathsf{tr}'_1.\mathsf{tr}'_2, \sigma'_{1/2})$.

We are now able to state our characterisation lemma. Intuitively, we show that for tests that are valid and guarded, it is sufficient to consider pulled-up tests. We first illustrate through an example how a test that is valid and guarded can be converted into a pulled-up one.

Example 7.3.3. Continuing Example 7.3.1, we consider the test $\mathsf{w}_7 = \mathsf{w}_9$ which is not pulled-up in $(\mathsf{tr}'_1.\mathsf{tr}'_2, \sigma'_{1/2})$. Consider the execution

$$\mathsf{tr}' = \mathsf{tr}'_1.\mathsf{io}(c_4, \mathsf{w}_5, \mathsf{w}_6).\mathsf{io}(c_5, \mathsf{w}_6, \mathsf{w}_7).\mathsf{io}(c_3, \mathsf{w}_4, \mathsf{w}_8).\mathsf{io}(c_4, \mathsf{w}_8, \mathsf{w}_9).\mathsf{io}(c_5, \mathsf{w}_9, \mathsf{w}_{10}).$$

This execution is almost similar to $\mathsf{tr}'_1.\mathsf{tr}'_2$. The main difference is that the computation performed at the end of tr'_1 using channel c_3 with input w_4 is duplicated. Both $\mathsf{io}(c_3, \mathsf{w}_4, \mathsf{w}_5)$ and $\mathsf{io}(c_3, \mathsf{w}_4, \mathsf{w}_8)$ occur in tr'. The resulting frame is:

$$\begin{array}{rcl} \sigma'_1 & \cup & \{\mathsf{w}_6 \triangleright \mathsf{senc}(\mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4), \mathsf{k}_2, r_7), \mathsf{k}_1, r_8),\ \mathsf{w}_7 \triangleright \mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4), \\ & & \mathsf{w}_8 \triangleright \mathsf{senc}(\mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4), \mathsf{k}_1, r_5), \mathsf{k}_2, r'_6), \\ & & \mathsf{w}_9 \triangleright \mathsf{senc}(\mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4), \mathsf{k}_2, r'_7), \mathsf{k}_1, r'_8),\ \mathsf{w}_{10} \triangleright \mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4)\}. \end{array}$$

The terms stored in w_5 and w_8 differ by their random seeds:

$$\mathsf{senc}(\mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4), \mathsf{k}_1, r_5), \mathsf{k}_2, r_6) \text{ and } \mathsf{senc}(\mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4), \mathsf{k}_1, r_5), \mathsf{k}_2, r'_6)$$

This frame is almost the same as $\sigma'_{1/2}$ with an additional element (w_8). The term stored in w_8 is the same as the one stored in w_5 up to the choice of some random seeds (r_6 is replaced by the fresh random r'_6). Moreover, the presence of this additional element leads us to reindex the following elements of the frame, and to replace some occurrences of r_6 with r'_6. It is important to note that the introduced randoms r'_6 and r'_8 could potentially break equality tests. They however do not appear anymore in the last outputted term stored in w_{10} that is checked for equality.

This example shows that when considering the trace $(\mathsf{tr}'_1.\mathsf{tr}'_2, \sigma'_{1/2})$, we may have to consider the test $\mathsf{w}_7 = \mathsf{w}_9$ which is not pulled-up. However, this test is essentially the same than the pulled-up test $\mathsf{w}_7 = \mathsf{w}_{10}$ issued from the trace given above.

The transformation explained in the previous example can be generalised to any protocol.

Lemma 7.3.1. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$, then $P \approx_{\mathsf{fwd}} Q$ if, and only if, the following four conditions are satisfied:

- CONST$_P$: For all $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$, there exists a frame σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$ and for every $\mathsf{w}, \mathsf{w}' \in dom(\sigma_P)$ and for every constant $\mathsf{c} \in \Sigma_0 \cup \{\mathsf{start}\}$, $\mathsf{w}\sigma_P = \mathsf{w}'\sigma_Q = \mathsf{c}$ if, and only if, there exists a constant $\mathsf{c}' \in \Sigma_0 \cup \{\mathsf{start}\}$ such that $\mathsf{w}\sigma_Q = \mathsf{w}'\sigma_Q = \mathsf{c}'$.
- CONST$_Q$: Similarly swapping the roles of P and Q.
- GUARDED$_P$: For all $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$, there exists a frame σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$ and every test that is σ_P-valid, σ_P-guarded, and pulled-up in (tr, σ_P) is also σ_Q-valid, σ_Q-guarded, and pulled-up in (tr, σ_Q).
- GUARDED$_Q$: Similarly swapping the roles of P and Q.

Proof. (sketch)

($\Rightarrow$) For this direction, when considering CONST$_P$, the only difficulty is to show that the test $\mathsf{w}\sigma_Q = \mathsf{w}'\sigma_Q$ leads to a constant c'. Actually, such a test can not lead to a guarded test since otherwise a replay of the entire sequence (this replay is possible since we consider a class of protocol that allows this) will lead to a different guarded term in Q and not in P (due to the presence of fresh randoms in guarded terms).

When considering GUARDED$_P$, the difficulty is to show that the test $\mathsf{w} = \mathsf{w}'$ is necessarily pulled-up in (tr, σ_Q). Let $\mathsf{pref} = \mathsf{io}(c_{i_0}, \mathsf{start}, \mathsf{w}_{j_0}) \ldots \mathsf{io}(c_{i_p}, \mathsf{w}_{j_{p-1}}, \mathsf{w}_{j_p})$ be the maximal common prefix of $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$. Since

$\mathsf{w} = \mathsf{w}'$ is pulled-up in (tr, σ_P), we know that the first occurrence of $\mathsf{w}\sigma_P$ in $\mathsf{pref}\sigma_P$ (if any) is at the very end of the sequence. We can easily show that $\mathsf{w} = \mathsf{w}'$ is σ_Q-valid and σ_Q-guarded, and thus $\mathsf{w}\sigma_Q$ occurs also as a subterm in $\mathsf{pref}\sigma_Q$. The only problem is if $\mathsf{w}\sigma_Q$ occurs in $\mathsf{pref}\sigma_Q$ but not at the very end of this sequence. The idea is that in such a case, we can modify the trace (tr, σ_Q) and the test $\mathsf{w} = \mathsf{w}'$ to build $(\mathsf{tr}^*, \sigma_Q^*)$ and a new test $\mathsf{w}_* = \mathsf{w}'_*$ which will be pulled-up in $(\mathsf{tr}^*, \sigma_Q^*)$. The idea is to split the two sequences $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$ earlier without compromising the fact that the test will be valid in the resulting frame. This corresponds to the construction illustrated in Example 7.3.3. This trace tr^* is actually a witness of non-equivalence. Actually, the test $\mathsf{w}_* = \mathsf{w}'_*$ is *a fortiori* not valid on the P side, and this contradicts our hypothesis $P \approx_{\mathsf{fwd}} Q$.

($\Leftarrow$) Actually, for this direction, assume that we have a witness of the fact that $P \not\approx Q$, *i.e.* a trace $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$, a trace $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$, and a test $\mathsf{w} = \mathsf{w}'$ that is σ_P-valid but not σ_Q-valid. In case the resulting term is a constant, we easily conclude that $\textsc{Const}_P$ fails. Otherwise, it means that $\mathsf{w} = \mathsf{w}'$ is σ_P-guarded. In order to show that $\textsc{Guarded}_P$ fails, we have to ensure that the test $\mathsf{w} = \mathsf{w}'$ is pulled-up w.r.t. (tr, σ_P). Since, this is not necessarily the case, we have to build another trace $(\mathsf{tr}^*, \sigma_P^*)$ that will lead us to a pulled-up test. Roughly, the transformation consists in splitting the two sequences $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$ earlier without compromising the fact that the test will be valid in the resulting frame. Actually, such a transformation can not transform a test that was not valid in a valid one, thus this test is still not valid for Q and it is still a witness of non-equivalence, but a pulled-up one allowing us to conclude. □

The detailed proof can be found in Appendix B.3.1.

7.3.3 From trace equivalence to language equivalence

Our goal is to associate an automaton $\mathcal{A}_P$ to a protocol P such that $\mathcal{A}_P$ recognises the words (a sequence of channels) that correspond to a possible execution of the protocol. The stack of the automaton $\mathcal{A}_P$ is used to store a (partial) representation of the last outputted term. This first requires to convert a term into a word.

Given an input term or an output term u (see Section 7.1.2), we define inductively $\bar{u}$ in the following way:

$$\begin{cases} \bar{u} = \bar{v}.\mathsf{k} & \text{if } u = \mathsf{f}(v, \mathsf{k}, r) \text{ and } \mathsf{f} \in \{\mathsf{senc}, \mathsf{raenc}, \mathsf{sign}\} \\ \bar{\mathsf{c}} = \omega\mathsf{c} & \text{for any constant } \mathsf{c} \in \Sigma_0 \cup \{\mathsf{start}\} \\ \bar{x} = \epsilon & \text{for any variable } x \end{cases}$$

where ϵ denotes the empty word. Note that, using this representation, random seeds are not part of the encoding. We denote by $||u||$ the *height* of the term u which is equal to the number of occurrence of senc, raenc, and sign in u.

We now consider an arbitrary ping-pong protocol P (using the same notation as the one introduced in Section 7.1):

$$P \stackrel{\mathsf{def}}{=} \prod_{i=1}^{n} \prod_{j=1}^{p_i} !\mathsf{in}(c_i, u_i^j).\mathsf{new}\ r_1.\ldots.\mathsf{new}\ r_{k_i^j}.\,\mathsf{out}(c_i, v_i^j) \qquad (*)$$

In the remaining of the section, we denote by Σ_0^P the finite set of constants of $\Sigma_0 \cup \{\mathsf{start}\}$ that actually occur in the protocol P.

Encoding of the conditions $\textsc{Const}_P$ and $\textsc{Const}_Q$

We first build an automaton that recognises tests of the form $\mathsf{w} = \mathsf{w}'$ such that the corresponding term is actually a constant. We define $\mathcal{A}^P_{\textsc{Const}}$ as follows:

$$\mathcal{A}^P_{\textsc{Const}} = (\{q_0, q_{\mathsf{f}}\} \cup \{q_{\mathsf{c}} \mid \mathsf{c} \in \Sigma_0^P\}, \{c_1, \ldots, c_n\} \cup \{c_{\mathsf{test}}, c_{\mathsf{end}}\}, \Sigma_0^P, q_0, \omega, \{q_0, q_{\mathsf{f}}\}, \delta)$$

where the transition function δ is defined as follows:

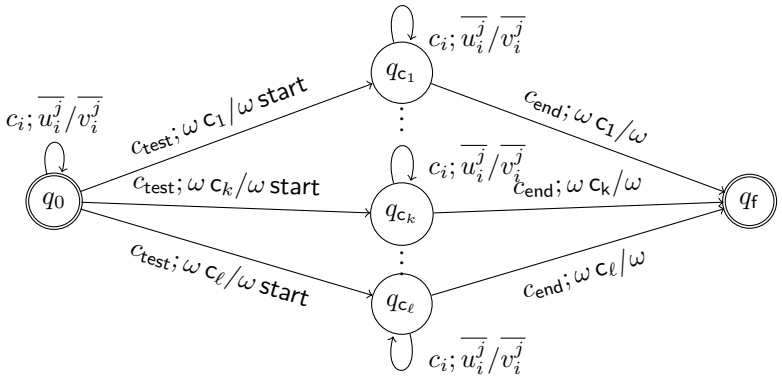


Figure 7.3: Automaton $\mathcal{A}^P_{\text{CONST}}$

1. for every $q \in \{q_0\} \cup \{q_{\mathsf{c}} \mid \mathsf{c} \in \Sigma_0^P\}$, for every $i \in \{1, \ldots, n\}$, for every $j \in \{1, \ldots, p_i\}$, there is a transition $q \xrightarrow{c_i; \bar{u}_j^i/\overline{v_j^i}} q$;
2. for every constant c, there is a transition $q_0 \xrightarrow{c_{\mathsf{test}}; \omega\ \mathsf{c}/\omega\ \mathsf{start}} q_{\mathsf{c}}$;
3. for every constant c, there is a transition $q_{\mathsf{c}} \xrightarrow{c_{\mathsf{end}}; \omega\ \mathsf{c}/\omega} q_{\mathsf{f}}$.

The automaton is depicted in Figure 7.3. Intuitively, the basic building blocks (e.g. q_0 with the transitions from q_0 to itself) mimic an execution of P where each input is fed with the last outputted term. Then, to recognise the tests of the form $\mathsf{w} = \mathsf{w}'$ that are true in such an execution, it is sufficient to memorise the constant c that is associated to w (adding a new state q_{c}), and to see whether it is possible to reach a state where the stack contains c again. More formally, we have the following result.

Lemma 7.3.2. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$, the two real-time GPDA $\mathcal{A}^P_{\text{CONST}}$ and A^Q_{CONST} are such that:

$$P \text{ and } Q \text{ satisfy conditions } \text{CONST}_P \text{ and } \text{CONST}_Q \text{ iff } \mathcal{L}(\mathcal{A}^P_{\text{CONST}}) = \mathcal{L}(\mathcal{A}^Q_{\text{CONST}}).$$

The proof can be found in Appendix B.3.2.

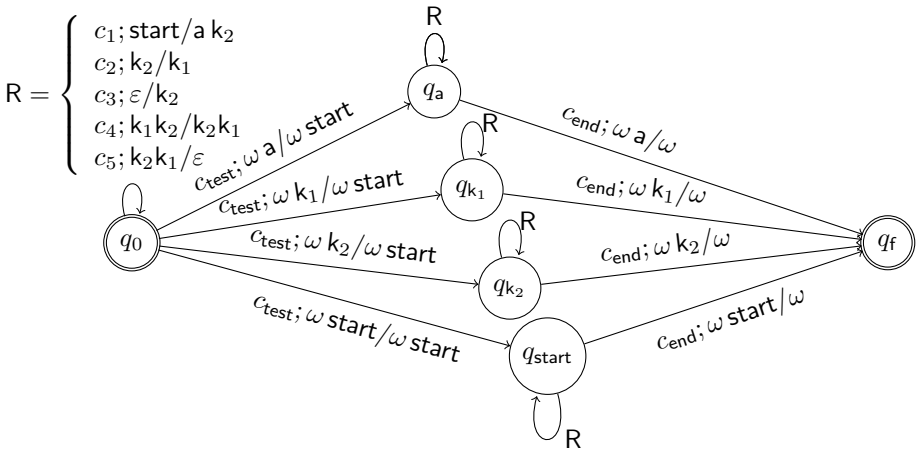
Example 7.3.4. Going back to our running example, *i.e.* the protocol P described in Figure 7.2, the automaton $\mathcal{A}^P_{\text{CONST}}$ is depicted below:

The word that represents the trace $(\mathsf{tr}_1.\mathsf{tr}_2, \sigma_{1/2})$ and the test $\mathsf{w}_5 = \mathsf{w}_8$ as given in Figure 7.2 is: $c_1c_2c_3c_4c_5c_{\mathsf{test}}c_1c_3c_2c_5c_{\mathsf{end}}$. The fact that this test is a valid one that leads to a constant a means that the word will be accepted by the automaton given above. The corresponding run goes through the state q_{a} and halts in state q_{f}.

$\mathcal{A}^P_{\text{CONST}}$ has a number of states polynomial in the number of constants in P, and for each state a number of transitions linear in the number of branches in P. Thus, $\mathcal{A}^P_{\text{CONST}}$ is of size polynomial with respect to the size of P.

Encoding of the conditions GUARDED_P and GUARDED_Q

Capturing tests that lead to non-constant symbols (*i.e.* terms of the form $\mathsf{f}(u, \mathsf{k}, r)$ with $\mathsf{f} \in \{\mathsf{senc}, \mathsf{raenc}, \mathsf{sign}\}$) is more tricky for several reasons. First, it is not possible anymore to memorise the resulting term in a state of the automaton. Second, names of sort rand play a role in such a test, while they are forgotten in our encoding. We rely on



our characterisation introduced in Section 7.3.2 and we construct a more complex automaton that uses some special track symbols to encode when randomised ciphertexts may be reused.

More precisely, we consider:

- $\Pi = \{c_1, \ldots, c_n, c_{\mathsf{test}}, c_{\mathsf{end}}\} \cup \{c^i_{\mathsf{fork}} \mid 1 \leq i \leq n\}$, and
- $\Gamma = \Sigma_0^P \cup \{\mathsf{test}\} \cup \{(\mathsf{fork}^j_i, k) \mid 1 \leq i \leq n,\ 1 \leq j \leq p_i,\ \text{and } 1 \leq k \leq ||u^j_i||\}$.

Note that n and p_i are induced by the definition of protocol P (see equation $(*)$). The input alphabet contains the channel names $c_1, \ldots, c_n$, plus some additional symbols, denoted $c^1_{\mathsf{fork}}, \ldots, c^n_{\mathsf{fork}}$, that will be used once and whose purpose will be to mark the end of the common prefix between $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$.

The stack-alphabet is more involved. We still have one symbol per constant in Σ_0^P, and a special symbol test that will be put on top of the stack when the stack contains the target term (*i.e.* $\mathsf{w}\sigma$). In such an automaton, the idea is to consider pulled-up tests only. The tile (fork^j_i, k) is placed on the stack when the automaton has finished to build the term corresponding to the left hand side of a pulled-up test.

The transition function δ is defined as follows:

1. for $q \in \{q_0, q_1, q_2\}$, for every $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, p_i\}$, there is a transition $q \xrightarrow{c_i;\, \overline{u^j_i}/\overline{v^j_i}} q$;
2. for every $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, p_i\}$ such that $||v^j_i|| \geq 1$, there is a transition
$$q_0 \xrightarrow{c^i_{\mathsf{fork}};\, \overline{u^j_i}/\omega\, \overline{v^j_i}\, (\mathsf{fork}^j_i, ||v^j_i||)} q_1$$
3. for every $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, p_i\}$, for every $i' \in \{1, \ldots, n\}$ and $j' \in \{1, \ldots, p_{i'}\}$, for every m such that $1 < m \leq ||v^{j'}_{i'}||$, and for every subterm u_0 of u^j_i of height $k \in \{1, \ldots, m-1\}$ such that $\overline{u^j_i} = \overline{u_0}.s'$ there is a transition
$$q_1 \xrightarrow{c_i;\, \overline{u_0}.(\mathsf{fork}^{j'}_{i'}, m).s'/(\mathsf{fork}^{j'}_{i'}, m-k)\overline{v^j_i}} q_1.$$
4. for every $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, p_i\}$, for every m such that $1 \leq m \leq ||v^j_i||$, there is a transition $q_1 \xrightarrow{c_{\mathsf{test}};\, (\mathsf{fork}^j_i, m)/\mathsf{test}\, s^j_{i,m}} q_2$ where $s^j_{i,m}$ is the suffix of length $||v^j_i|| - m$ of $\overline{v^j_i}$.
5. there is a transition $q_2 \xrightarrow{c_{\mathsf{end}};\, \mathsf{test}/\epsilon} q_{\mathsf{f}}$.

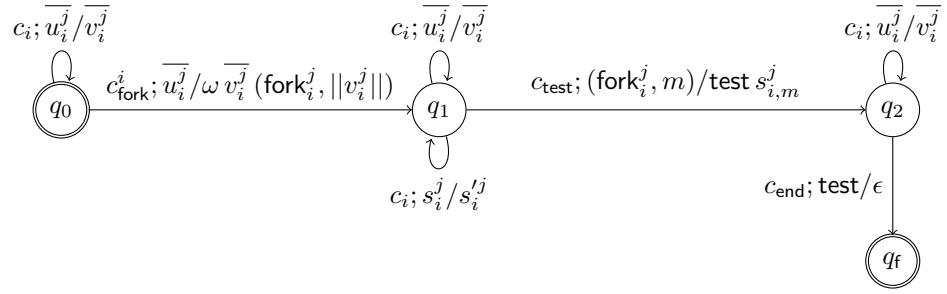


Figure 7.4: Automaton $\mathcal{A}^P_{\text{GUARDED}}$

The loop in q_0 (item 1) represents the regular execution of the protocol by the attacker: through unstacking and stacking, she builds a term on the stack along a particular trace. The transitions $q_0 \xrightarrow{c^i_{\mathsf{fork}};\, z/z'} q_1$ (item 2) enable her to mark a fork when building a test in her frame with a particular stack symbol fork_i^j, enriched with some information. Intuitively, the part of the execution that is performed until here should correspond to the maximal prefix shared between the sequences $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$. By looping in q_1, the attacker can continue building the first term of an equality, following the usual execution of the protocol, if it were for the presence of the stack symbol (fork_i^j, k) which can only go down on the stack for at most $k-1$ times. When the symbol (fork_i^j, k) appears on top of the stack, the attacker may decide that she has built the first part of a pulled-up test. Then test will be put on the top of the stack and a part of the stack (following the instructions memorised in the symbol (fork_i^j, k)) will be regenerated. The idea is that the stack has to contain the same term as the one stored just after forking. Then the attacker tries to build the second member of the test. If this second term manages to end up exactly as the previous one (the position in the stack is marked using the tile test), an equality is reached and the word is recognised by the automata, witnessing the equality induced by the pulled-up test.

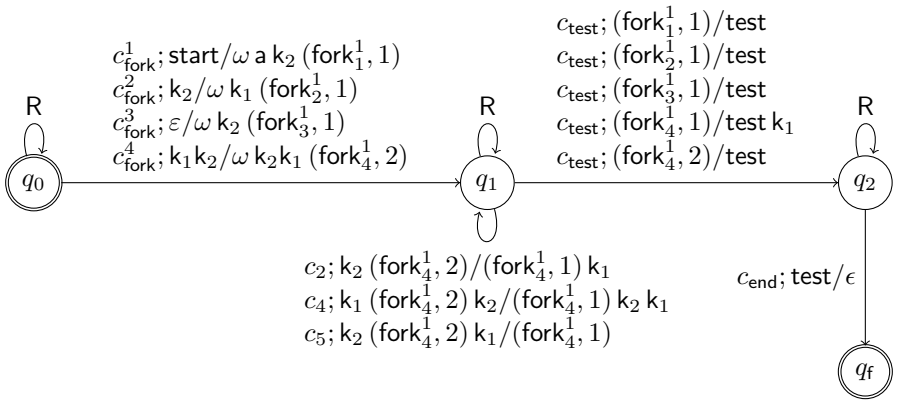
What remains now is to prove that P and Q satisfy conditions GUARDED_P and GUARDED_Q if, and only if, $\mathcal{L}(\mathcal{A}^P_{\text{GUARDED}}) = \mathcal{L}(\mathcal{A}^Q_{\text{GUARDED}})$. This is formally stated in the following lemma.

Lemma 7.3.3. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$, the two real-time GPDA $\mathcal{A}^P_{\text{GUARDED}}$ and A^Q_{GUARDED} are such that:

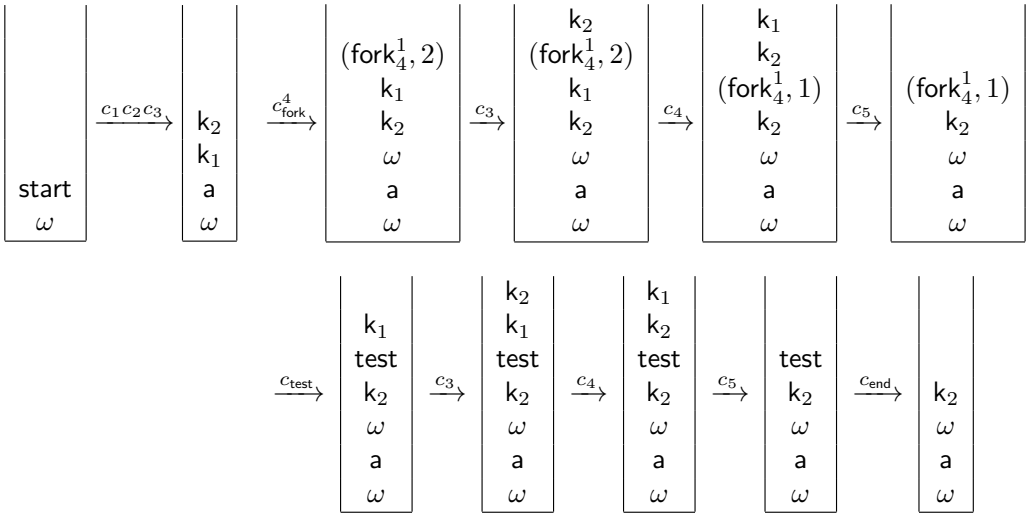
$$P \text{ and } Q \text{ satisfy conditions } \text{GUARDED}_P \text{ and } \text{GUARDED}_Q \text{ iff } \mathcal{L}(\mathcal{A}^P_{\text{GUARDED}}) = \mathcal{L}(\mathcal{A}^Q_{\text{GUARDED}}).$$

The proof can be found in Appendix B.3.2.

Example 7.3.5. Going back to our running example, *i.e.* the protocol P described in Figure 7.2, the automaton $\mathcal{A}^P_{\text{GUARDED}}$ is depicted below.



The set of transitions R is the one defined in Example 7.3.4. The situation where the stack symbol (fork^j_i, k) goes down occurs for instance when considering the word $c_1\, c_2\, c_3\, c^4_{\mathsf{fork}}\, c_3\, c_4\, c_5\, c_{\mathsf{test}}\, c_3\, c_4\, c_5$. The evolution of the stack during the run of the automaton is depicted below. On the second line, we can see that this symbol goes down and k goes from 2 to 1.



The trace $(\mathsf{tr}, \sigma) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ and the pulled-up test $\mathsf{w} = \mathsf{w}'$ that correspond to this execution is the ones introduced in Example 7.3.3, *i.e.* tr' together with the test $\mathsf{w}_7 = \mathsf{w}_{10}$.

We can notice that up to the special stack-symbols, namely test and (fork^j_i, k), the contents of the stack after reading c^i_{fork} (here $i = 4$) and c_{test} are the same. The stack actually represents the term obtained after executing the common prefix shared between $\mathsf{seq}_{\mathsf{tr}'}(\mathsf{w}_7)$ and $\mathsf{seq}_{\mathsf{tr}'}(\mathsf{w}_{10})$, *i.e.* $\mathsf{senc}(\mathsf{senc}(\mathsf{a}, \mathsf{k}_2, r_4), \mathsf{k}_1, r_5)$ stored in w_4. We have also that the contents of the stack before reading c_{test} and after reading c_{end} are also the same (up to some special stack symbols). They actually represent the terms stored respectively in w_7 and w_{10}.

Note that $\mathcal{A}^P_{\text{GUARDED}}$ has a fixed number of states, and a polynomial number of transitions : transitions are added for each branch and suffix of any input term. Thus, $\mathcal{A}^P_{\text{GUARDED}}$ is of size polynomial with respect to the size of P.

7.4 From language equivalence to trace equivalence

We have seen how to encode trace equivalence between processes in $\mathcal{C}_{\mathsf{pp}}$ into language equivalence between real-time GPDA. The two problems are actually *equivalent*. Indeed, in this section, we show that we can conversely encode any

real-time GPDA $\mathcal{A}$ into a process $P_{\mathcal{A}}$ in $\mathcal{C}_{\mathsf{pp}}$ such that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ implies $P_{\mathcal{A}} \sqsubseteq P_{\mathcal{B}}$.

Consider an automaton $\mathcal{A} = (Q, \Pi, \Gamma, q_0, \omega, Q_f, \delta)$. The process $P_{\mathcal{A}}$ associated to $\mathcal{A}$ is built using symmetric encryption only. For the purpose of the encoding, we consider the following constants of sort SymKey:

- for each $q \in Q$, we denote q its counterpart in Σ_0;
- for each $\alpha \in \Gamma$, we denote k_α its counterpart in Σ_0;
- a constant $\mathsf{k}_{\mathsf{well}}$.

Let also c_0, c_a, c_f with $a \in \Pi$ be constant symbols of sort $\mathsf{channel}$ in $\mathcal{C}h$. Words in Γ^*, *i.e.* stacks, will be represented through nested encryptions with keys representing their counterparts in Γ. For the sake of brevity, given a word $u = \alpha_1 \dots \alpha_p$ of Γ^*, we denote by $\overline{x.u}$:

- either the term $\mathsf{senc}(\dots \mathsf{senc}(x, \mathsf{k}_{\alpha_1}, z_1) \dots, \mathsf{k}_{\alpha_p}, z_p)$ where z_1 through z_p are variables used for nonces when $\overline{x.u}$ is used in as an input pattern;
- or the term $\mathsf{senc}(\dots \mathsf{senc}(x, \mathsf{k}_{\alpha_1}, r_1) \dots, \mathsf{k}_{\alpha_p}, r_p)$ where r_1 through r_p are fresh randoms when $\overline{x.u}$ is used as an output pattern.

Below, we use $\mathsf{new}\, \tilde{r}$ as shortcut for $\mathsf{new}\, r_1. \dots \mathsf{new}\, r_p$ such that the sequence will bind every nonce occurring in the following output.

The stack of the automaton $\mathcal{A}$ is encoded as a pile of encryptions (where each key encodes a letter of the stack). Then, upon receiving such a pile of encryptions encrypted by q on channel c_a, the process $P_{\mathcal{A}}$ will mimic the transition of $\mathcal{A}$ that is triggered when the automaton is at state q upon reading a with the stack corresponding to that pile of encryptions.

More formally, the process $P_{\mathcal{A}}$ is defined as follows:

$$
\begin{array}{rll}
P_{\mathcal{A}} \stackrel{\mathsf{def}}{=} & !\,\mathsf{in}(c_0, \mathsf{start}).\mathsf{new}\ \tilde{r}.\mathsf{out}(c_0, \mathsf{enc}(\mathsf{enc}(\mathsf{start}, \mathsf{k}_\omega, r_1), \mathsf{k}_{\mathsf{start}}, r_2), q_0, r_3)) & (0) \\
| & !\,\mathsf{in}(c_a, \mathsf{enc}(\overline{x.u}, q, z)).\mathsf{new}\ \tilde{r}.\mathsf{out}(c_a, \mathsf{enc}(\overline{x.v}, q', r)) & (1) \\
| & !\,\mathsf{in}(c_a, \mathsf{enc}(\overline{x.u'}, q, z)).\mathsf{new}\ r.\mathsf{out}(c_a, \mathsf{enc}(\mathsf{start}, \mathsf{k}_{\mathsf{well}}, r)) & (1a) \\
| & !\,\mathsf{in}(c_a, \mathsf{enc}(\mathsf{start}, \mathsf{k}_{\mathsf{well}}, z)).\mathsf{new}\ r.\mathsf{out}(c_a, \mathsf{enc}(\mathsf{start}, \mathsf{k}_{\mathsf{well}}, r)) & (1b) \\
| & !\,\mathsf{in}(c_f, \mathsf{enc}(x, q_f, z)).\mathsf{new}\ r.\mathsf{out}(c_f, \mathsf{enc}(\mathsf{start}, q_f, r)) & (2)
\end{array}
$$

where a quantifies over Π, q over Q, u over words in Γ^* such that $(q, a, u) \in dom(\delta)$, q_f over Q_f, and $(q', v) = \delta(q, a, u)$. Lastly, u' ranges over $U'_{q,a} \stackrel{\mathsf{def}}{=} \alpha \cdot SS_{q,a} \setminus S_{q,a}$ where $S_{q,a}$ (resp. $SS_{q,a}$) is the set that contains suffixes (resp. strict suffixes) of some u with $(q, a, u) \in dom(\delta)$. This set $U'_{q,a}$ corresponds intuitively to the set of shortest words which are not suffixes of any word in $\{u \mid (q, a, u) \in dom(\delta)\}$, and, thus the shortest words to unstack to be sure that no transition from q reading a is possible in the automaton.

Example 7.4.1. Consider a real-time GPDA such that $\Gamma = \{\alpha, \beta, \gamma, \omega\}$, $q \in Q$, and $a \in \Pi$. Assume that $\{u \mid (q, a, u) \in dom(\delta)\} = \{\beta\alpha,\ \beta\alpha\alpha\}$. We have $SS_{q,a} = \{\epsilon, \alpha, \alpha\alpha\}$, and $S_{q,a} = SS_{q,a} \cup \{\beta\alpha, \beta\alpha\alpha\}$. Thus, we have that:

$$U'_{q,a} = \{\omega, \beta, \gamma, \omega\alpha, \gamma\alpha, \omega\alpha\alpha, \alpha\alpha\alpha, \gamma\alpha\alpha\}.$$

In the encoding above, the branches (0) and (1) mimic the behaviour of the automaton $\mathcal{A}$. Branch (2) is triggered in case a final state q_f is reached. In case we are considering a behaviour that is not authorised by the automaton, we obtain a message encrypted with $\mathsf{k}_{\mathsf{well}}$ through branches $(1a)$. Then branches $(1b)$ allow to pursue the execution of the protocol outputting messages that look fresh.

Lemma 7.4.1. The protocol $P_{\mathcal{A}}$ described above is in $\mathcal{C}_{\mathsf{pp}}$ and of size polynomial w.r.t. $\mathcal{A}$.

Proof. First, note that because $dom(\delta)$ is finite, as the automaton is finitely described, the sets $\{u \mid (q, a, u) \in dom(\delta)\}$ and $U'_{q,a}$ are also finite for any $a \in \Pi$ and $q \in Q$. Moreover, the automaton being deterministic, given $q \in Q$ and $a \in \Pi$, for every word $s \in \Gamma^*$:

- either there exists a unique suffix u of s such that $(q, a, u) \in dom(\delta)$;
- or there exists a unique suffix u' of s such that $u' \in U'_{q,a}$,

and this disjunction is exclusive. This allows us to ensure that condition (2) of Definition 7.1.1 is satisfied, and thus $P_{\mathcal{A}}$ belongs to $\mathcal{C}_{\mathsf{pp}}$.

Regarding the size of the protocol, the only non-trivial point is to check that the number of branches $(1a)$ is polynomially bounded. Let $q \in Q$, and $a \in \Pi$, and assume that the maximal length of a word u in a transition $q \xrightarrow{a;u/v} q'$ of the automaton is $\ell_{q,a}$, we have that the number of branches $(1a)$ for state q and letter a is bounded by $\ell_{q,a} \times \#\Gamma \times \#\{u \mid (q, a, u) \in dom(\delta)\}$ where $\#S$ is the cardinality of set S. This allows us to conclude. □

This polynomial encoding preserves inclusion.

Proposition 7.4.1. Let $\mathcal{A}$ and $\mathcal{B}$ be two real-time GPDA. We have that:

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B}) \iff P_{\mathcal{A}} \sqsubseteq P_{\mathcal{B}}.$$

Proof. Let $\mathcal{A} = (Q, \Pi, \Gamma, q_0, \omega, Q_f, \delta)$ and $\mathcal{B} = (Q', \Pi, \Gamma', q'_0, \omega, Q'_f, \delta')$. We show the two implications separately.
($\Leftarrow$) Assume that there exists a word $t \in \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{B})$. We will build a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P_{\mathcal{A}})$ such that there exists no trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(P_{\mathcal{B}})$ allowing us to conclude that $P_A \not\sqsubseteq P_{\mathcal{B}}$. To build (tr, ϕ), we will mimic the behaviour of $\mathcal{A}$ when reading t. The first branch to use is (0), enabling the attacker to activate other branches of the process $P_{\mathcal{A}}$. As $t \in \mathcal{L}(\mathcal{A})$ and $\mathcal{A}$ is deterministic, there exists a unique sequence of transitions leading to an accepting state $q_f \in Q_f$. For every such transition the attacker will activate the corresponding branch (1) in $P_{\mathcal{A}}$. If $t = a_1 \dots a_n$, we define (tr, ϕ) as follows:

$$\mathsf{tr} = \quad \mathsf{io}(c_0, \mathsf{start}, \mathsf{w}_1).\mathsf{io}(c_{a_1}, \mathsf{w}_1, \mathsf{w}_2) \dots \mathsf{io}(c_{a_n}, \mathsf{w}_n, \mathsf{w}_{n+1}).\mathsf{io}(c_f, \mathsf{w}_{n+1}, \mathsf{w}_{n+2})$$

and ϕ is defined as expected given our semantics. Because of the definition of the branch (1), the inputs on the channels c_{a_i} are possible, the stack of the automaton upon reading a_i and its current state being faithfully represented by the term $\mathsf{w}_i\phi$. Thus, (tr, ϕ) is indeed a trace of $P_{\mathcal{A}}$. When reaching q_f, the attacker can use the branch (2) and output the message $\mathsf{senc}(\mathsf{start}, q_f, r)$. As $t \notin \mathcal{L}(\mathcal{B})$, the corresponding sequence of transitions in $\mathcal{B}$ does not lead to any accepting state:

- either at some point of the execution of the automaton a transition from state q reading a is not possible with the current stack s. This means that there does not exist a suffix u of s such that $(q, a, u) \in dom(\delta')$, and thus, by definition of $U'_{q,a}$, there exists a suffix u' of s such that $u' \in U'_{q,a}$, enabling a transition $(1a)$ on channel c_a for the attacker, and every subsequent transition is done using branches $(1b)$,
- or the state reached in $\mathcal{B}$ after reading t is not an accepting state, *i.e.* not in Q'_f: the sequence $\mathsf{in}(c_f, \mathsf{w}_{n+1}).\mathsf{out}(c_f, \mathsf{w}_{n+2})$ cannot occur in $P_{\mathcal{B}}$.

Consequently, there exists no trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(P_{\mathcal{B}})$ (for any ψ), thus $P_{\mathcal{A}} \not\sqsubseteq P_{\mathcal{B}}$.

($\Rightarrow$) First note that, for every frame ϕ (resp ψ) such that $(\mathsf{tr}, \phi) \in \mathsf{trace}(P_{\mathcal{A}})$ (resp. $(\mathsf{tr}, \psi) \in \mathsf{trace}(P_{\mathcal{B}})$), we have that ϕ (resp. ψ) is of the form

$$\{\mathsf{w}_1 \triangleright \mathsf{senc}(m_1, k_1, r_1), \dots, \mathsf{w}_n \triangleright \mathsf{senc}(m_n, k_n, r_n)\}$$

where the k_i are non deducible and the r_i are "fresh" in the sense that they are all distinct and non deducible. This means that no equality (but the trivial ones) holds in such a frame. Now consider the shortest trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P_{\mathcal{A}})$, in terms of number of transitions, such that there exists no equivalent frame $(\mathsf{tr}, \psi) \in \mathsf{trace}(P_{\mathcal{B}})$. Since keys are non-deducible, we may assume w.l.o.g that $(\mathsf{tr}, \phi) \in \mathsf{trace}_{\mathsf{fwd}}(P_{\mathcal{A}})$. Because of the branches (1), $(1a)$ and $(1b)$ and in particular of the definition of $U'_{q,a}$, for any $q \in Q$, for any $a \in \Pi$, a transition of channel c_a is always possible, and we have seen that the resulting frames are necessarily in static equivalence. Thus, the only shortest trace where $P_{\mathcal{B}}$ will not be able to follow is when tr ends with an input/output on channel c_f. Let $\mathsf{w} \in dom(\phi)$ be the corresponding variable in the frame ϕ. Consider the subsequence $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ of tr and more precisely the sequence of channels that occurs in this subsequence. Such a sequence is of the form: $c_0.c_{a_1} \dots c_{a_n}.c_f$.

Let $v = a_1 \dots a_n$. We have that v is a word of Π^*, and, in particular,

- $v \in \mathcal{L}(\mathcal{A})$: indeed, branches (1) in $P_{\mathcal{A}}$ faithfully represent transitions $(q, a, u) \in dom(\delta)$ and a branch (2) can only be fired if $q_f \in Q_f$.
- $v \notin \mathcal{L}(\mathcal{B})$: indeed branch (2) could not be fired, either $\mathcal{B}$ cannot read v or, after reading v, $\mathcal{B}$ is not in any state of Q'_f.

Hence $v \in \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{B})$, proving that $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$. □

Therefore, checking for equivalence of protocols is as difficult as checking equivalence of real-time generalised pushdown deterministic automata.

7.5 Implementation

In this section, we detail our tool `Cpp2dpa` to convert protocols in $\mathcal{C}_{\mathsf{pp}}$ into GDPA, available online at

`http://www.lsv.ens-cachan.fr/~chretien/cpp2dpa.php`.

This tool takes two protocols in $\mathcal{C}_{\mathsf{pp}}$ as input, turn them into GDPA and, through the tool `lAlBlC` [51], outputs whether the two protocols were in equivalence, yielding a witness of non-equivalence in the negative case in the form of a sequence of channels leading to an attack. The tool focuses on the encoding as described in Section 7.3. In particular, we assume the prior steps of Section 7.2 were successfully applied to the pair of protocols; namely the bijection α as in Lemma 7.2.2 was successfully guessed and the oracles of Section 7.2.2 correctly added.

The tool `Cpp2dpa` is written in Python 3. From pairs of protocols in $\mathcal{C}_{\mathsf{pp}}$, it generates three pairs of *normalised deterministic pushdown automaton*, instead of directly two pairs of GPDA (as described in Section 7.3). This was necessary so as to interface with `lAlBlC`, and involves no loss of generality, as the former are more expressive than our GDPA. The normalisation process still has the inconvenient, in order to preserve the determinacy of the result, to output automata that may duplicate actions. More specifically, when necessary, the channels appearing in the potential witness of non-equivalence may be doubled. This technical detail does not impair the ability for the combined tool to prove equivalence or find witnesses, nevertheless.

7.5.1 Encoding pairs

Most protocols use pairs. While our formalism does not directly support pairs, we may encode a restricted kind of pairing, when there are only constants (such as identities) on the right. Formally, this amounts into encoding a pair $\langle t, a \rangle$, where t is a term and a some constant, by an encryption $\mathsf{senc}(t, a, r)$ for some random seed r. Provided constants used in concatenation are disjoint from constants used as keys, this encoding does not introduce any confusion. Note that since encryption is randomised, this pairing operator also differs as it is randomised.

7.5.2 Biometric passport

We are interested here in proving the unlinkability of the electronic passport protocol. A detailed specification of it can be found in [5]. Here, we only consider the passport's role and forget about the reader. The first case we consider is the flawed version corresponding to the French implementation of the passport, in which an attack arises from the ability for the attacker to observe whether a MAC check succeeds or not. As our framework does not directly enables us to deal with pairs of messages with their MAC, we model it by a signature: the attacker is able to obtain the plaintext of it (which amounts to retrieving the first component of the real pair) but cannot forge it (the attacker is not a priori able to forge a valid MAC). The resulting process is defined as follows.

$$\begin{aligned}
P_A \stackrel{\mathsf{def}}{=} & \,!\,\mathsf{in}(c_1, \mathsf{start}).\mathsf{new}\,\tilde{r}'. \\
& \quad \mathsf{out}(c_1, \mathsf{sign}(\mathsf{senc}(\mathsf{senc}(\mathsf{senc}(n_r, \mathsf{k}_r, r_1'), n_p, r_2'), \mathsf{k}_E, r_3'), \mathsf{mac}_{\mathsf{k}_m}, r_4')) \qquad (1) \\
& \mid !\,\mathsf{in}(c_2, \mathsf{sign}(\mathsf{senc}(x, \mathsf{k}_E, z_1), \mathsf{mac}_{\mathsf{k}_m}, z_2)).\mathsf{new}\, r_5.\mathsf{out}(c_2, \mathsf{sign}(x, \mathsf{mac}_{\mathsf{ok}}, r_5)) \qquad (2a) \\
& \mid !\,\mathsf{in}(c_2', \mathsf{sign}(\mathsf{senc}(x, n_p, z_1), \mathsf{mac}_{\mathsf{ok}}, z_2)).\mathsf{new}\,\tilde{r}''. \\
& \qquad\qquad \mathsf{out}(c_2', \mathsf{sign}(\mathsf{senc}(\mathsf{senc}(x, n_p, r_1''), \mathsf{k}_p, r_2''), \mathsf{mac}_{\mathsf{k}_m}, r_3'')) \qquad (2b)
\end{aligned}$$

where $\mathsf{new}\,\tilde{r}$ is a shortcut of $\mathsf{new}\, r_1.\mathsf{new}\, r_2.\mathsf{new}\, r_3.\mathsf{new}\, r_4$ (and similarly for $\mathsf{new}\,\tilde{r}'$ and $\mathsf{new}\,\tilde{r}''$). The protocol is modelled through three rules. Branch (1) corresponds to a message from the current session, emitted by the reader. While the original protocol can check the authenticity of the MAC and the value of the nonce sent to the passport, our formalism requires us to separate this into two steps: branches $(2a)$ and $(2b)$. Branch $(2a)$ checks the validity of the MAC: if it is, it send a message signed with $\mathsf{mac}_{\mathsf{ok}}$. On the other hand, branch $(2b)$ checks the value of the nonce (*i.e.* n_p) and finally emits the last message of the protocol. To retrieve the attack, we introduce the message sent by the reader from a previous session with a new branch denoted (0):

$$\begin{aligned}
& !\,\mathsf{in}(c_0, \mathsf{start}).\mathsf{new}\,\tilde{r}. \\
& \quad \mathsf{out}(c_0, \mathsf{sign}(\mathsf{senc}(\mathsf{senc}(\mathsf{senc}(n_r^0, \mathsf{k}_r^0, r_1), n_p^0, r_2), \mathsf{k}_E, r_3), \mathsf{mac}_{\mathsf{k}_m}, r_4)) \qquad (0)
\end{aligned}$$

Another protocol P_B is obtained by replacing $\mathsf{mac}_{\mathsf{k}_m}$ by $\mathsf{mac}_{\mathsf{k}_m'}$ in branches (1), $(2a)$ and $(2b)$. Our tool `Cpp2dpa` can automatically check that $P_A \not\approx P_B$.

Another version P_A' is obtained by replacing branches $(2a)$ and $(2b)$ by the branch

$$\begin{aligned}
& !\,\mathsf{in}(c_2, \mathsf{sign}(\mathsf{senc}(\mathsf{senc}(x, n_p, z_1), \mathsf{k}_E, z_2), \mathsf{mac}_{\mathsf{k}_m}, z_3)).\mathsf{new}\,\tilde{r}''. \\
& \qquad\qquad \mathsf{out}(c_2, \mathsf{sign}(\mathsf{senc}(\mathsf{senc}(x, n_p, r_1''), \mathsf{k}_E, r_2''), \mathsf{mac}_{\mathsf{k}_m}, r_3'')) \qquad (2)
\end{aligned}$$

The protocol P_B' is similarly defined, with $\mathsf{mac}_{\mathsf{k}_m'}$ instead of $\mathsf{mac}_{\mathsf{k}_m}$ in this branch (2). This version models the safe implementation of the protocol, where the success or failure of the MAC check is invisible to the attacker. Our tool `Cpp2dpa` can automatically prove that $P_A' \approx P_B'$.

7.5.3 Experiments

We have tested our tool `Cpp2dpa` on the running example as defined in Example 7.1.1 and Example 7.2.2; as well as on an encoding of the electronic passport protocol, described in Section 7.5.2 in two versions, unsafe and safe (see [5] for more details).

	Automata (in ms)	Grammars (in s)	Equivalence (in s)
Example 7.1.1	7.1	9.2	3462 (attack)
Example 7.2.2	7.0	3.1	9788 (proof)
Unsafe passport	7.1	23.2	4.89 (attack)
Safe passport	8.1	15.0	76.1 (proof)

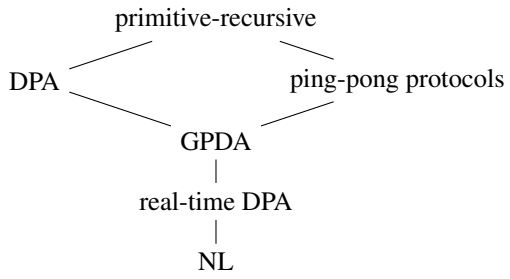


Figure 7.5: Complexity bounds for equivalence of ping-pong protocols.

The experiments were conducted on a Intel(R) Xeon(R) CPU X5650 @ 2.67GHz with 47 Go of RAM, using one core only. The first column corresponds to the cumulative time required to produce the different automata; the second one the time needed to convert the automata into grammars to be processed by `lAlBlC` and the third one to the status of the equivalence (proof or witness of non-equivalence) and the cumulative time spent to prove the equivalence (when it is the case) or the execution time to find a witness of non-equivalence, when possible. There is non-equivalence as soon as one of our three pairs of automata are not in equivalence. Since we execute `lAlBlC` in parallel for each of these three pairs, the execution time corresponds to the first pair that is found to be not in equivalence. Converting the automata to grammars required an optimisation of the built-in functionality in `lAlBlC` in order to reach reasonable execution times. Other protocols were considered, namely variants of the Wide Mouthed Frog, Denning-Sacco and Private Authentication protocols. Unfortunately, for these ones, albeit the generation of automata was quick, it was impossible to prove (non-)equivalence in reasonable time with `lAlBlC`.

7.6 Conclusion

We have shown a first decidability result for equivalence of (deterministic) ping-pong protocols for an unbounded number of sessions by reducing it to the equality of languages of deterministic, generalised, real-time pushdown automata (GPDA). We further show that deciding equivalence of ping-pong protocols is actually at least as hard as deciding equality of languages of GPDA. Complexity-wise, the situation is slightly less clear. While the reduction from GPDA to ping-pong protocols is polynomial, the reduction from ping-pong protocols to GPDA requires an exponential blow-up. Indeed, to get rid of the attacker, we guess a correspondence between the keys of P and Q, and exponentially many such correspondences should be checked. In addition, the complexity of equivalence of various classes of pushdown automata are not very well-known. It follows that the exact complexity of checking equivalence of protocols is unknown. The only upper bound is that equivalence is at most primitive recursive. This bound comes from the algorithm proposed by C. Stirling for equivalence of DPA [64]. The lower bound comes from the fact that real-time deterministic pushdown automata are at least NL-hard [16]. Whether equivalence of DPA (or even real-time GPDA) is e.g. at least NP-hard is unknown. The complexity hierarchy known so far for equivalence of ping-pong protocols is displayed in Figure 7.5.

Note that the complexity of GPDA and ping-pong protocols is actually quite close since the reduction from ping-pong protocols to GPDA is "just" exponential. Moreover, assume now that we consider only procedures that return a witness of non equivalence (if any). Then the complexity classes of GPDA and ping-pong protocols should actually coincide. Indeed, assume that there is a procedure for checking equivalence of GPDA that ends in time $f(n)$ where n is the size of the inputs, and that returns a witness when two automata are not in equivalence. This witness must be of size at most $f(n)$. Then given two ping-pong protocols P and Q, we would construct $\bar{P}$ and $\bar{Q}$ as defined in Lemma 7.2.2 step by step.

Instead of guessing the sets K and K', we would start from the empty sets $K = K' = \emptyset$. If $\bar{P} \not\approx \bar{Q}$, that is if

$\mathcal{A}_{\bar{P}} \not\approx \mathcal{A}_{\bar{Q}}$, we consider a witness of non equivalence. Either it is a witness of $P \not\approx Q$ (and we are done), or there must exist a key k that is deducible in P and a corresponding key k' deducible with the same actions in Q. We start over with $K = \{\mathsf{k}\}$ and $K' = \{\mathsf{k}'\}$.

This algorithm has at most n steps and each step involve a call to the GPDA procedure ($\mathcal{A}_{\bar{P}} \approx \mathcal{A}_{\bar{Q}}$) and involves replaying a witness of size $f(n)$. This yields a procedure of complexity $O(f(n))$.

Our class of security protocols handles only randomised primitives, namely symmetric/asymmetric encryptions and signatures. Our decidability result could be extended to handle deterministic primitives instead of the randomised one (the reverse encoding - from real-time GPDAs to processes with deterministic encryption - may not hold anymore). Due to the use of pushdown automata, extending our decidability result to protocols with pairing is not straightforward. A direction is to use pushdown automata for which stacks are terms.

Our tool `Cpp2dpa` in combination with `lAlBlC` yields the first implementation of a decidability procedure for equivalence of protocols, for an unbounded number of sessions. However, the number of protocols covered so far is limited. A first reason yields in the limitations of the class of ping-pong protocols. However, another reason is the (too long) time needed to check for equivalence. Our transformation from protocols to automata using `Cpp2dpa` remains reasonably fast. Most of the execution time comes from `lAlBlC`. Since this tool is still in its early stage of development, we may hope for significant improvement of `lAlBlC`' performance in the next years.

Chapter 8

Conclusion and perspectives

This thesis reports my contributions to the automated verification of trace equivalence for cryptographic protocols. In particular, we focused our efforts on the issue of deciding trace equivalence for *an unbounded number of sessions*. Doing so, we propose two ways of easing equivalence checking in general and several classes of cryptographic protocols for which trace equivalence can be decided. We recall here our main contributions to this topic and open further perspectives.

Easing trace equivalence with nonce deletion

Chapter 3 is devoted to the definition of a transformation on simple protocols using a broad variety of cryptographic primitives which allows us to soundly check for trace equivalence *without nonces*. By making sure any equivalence between such transformed protocols leads to an equivalence between the original protocols with nonces, we turn decidability results for trace equivalence for an unbounded number of sessions but without nonces, such as the one described in Chapter 5, into a sound terminating procedure to prove protocols with nonces. Even though the hypothesis on simple protocols is still practical, it may be possible to extend it further to encompass more semantic definitions of determinism, such as action-determinism [8]. Similarly, our definition of adequate theories focuses on theories with destructors and constructors but can potentially be extended to more arbitrary theories, to the cost of a slight generalisation of our transformation. More specifically, our transformation introduces one extra copy of each deleted nonce to account for the simple structure of the rewriting rules which describe adequate theories such as encryption. When dealing with more complex theories, adding more copies of the said nonce could allow us to extend our soundness result. The most promising extension to this work yet lies into its application to bounds of different natures. In the same way we bound the number of nonces to use when soundly checking trace equivalence, quantities like the number of agents interacting in a protocol may also be soundly bounded with our approach [38].

Easing trace equivalence with typing

Chapter 4 offers a new typing result for trace equivalence of determinate protocols with symmetric encryption. Akin to a previous result for reachability properties [6], it can be seen as an extension to trace equivalence with more general typing systems, although for a smaller set of primitives. This typing result offers an important reduction in the size of the search space to consider when checking for trace equivalence and provides the corner stone to the decidability results of both Chapter 5 and Chapter 6. As this result is proven by designing a new decision procedure to check equivalence for a bounded number of sessions which limits the amount of unification between terms by only considering terms of the exact same type, a further application would be to apply this optimisation to existing tools such as Apte [20] or Spec [65] to improve their performances. Another natural improvement is to extend the rewriting rules we considered to include other standard primitives like asymmetric encryption, signatures or hashes. To do so,

we need to redefine the procedure we designed for a bounded number of sessions. In particular, while symmetric encryption offers inversibility (as the encryption and decryption keys are identical) which greatly limits the amount of unification to do in our algorithm, other primitives such as asymmetric encryption or signatures lack it, forcing us to adapt this step of the algorithm. This extension would also most likely provide an extension to the decidability results in Chapters 5 and 6.

A new decidable class: simple type-compliant protocols without nonces

Chapter 5 presents one of the first class of protocols for which trace equivalence is decidable for an unbounded number of sessions. This class is made of simple, type-compliant protocols without nonces using symmetric encryption. Simple tagged protocols without nonces form an interesting subclass of these. Although forbidding arbitrary nonces seems unpractical, the result from Chapter 3 offers a practical and sound extension of this result to checking trace equivalence for an unbounded number of sessions with nonces. As mentioned earlier, this result relies deeply on the typing result from Chapter 4 and any extension of the original signature would likely carry over to the decidability result. Another axis of research is more closely related to our proof method: by bounding the nonces and bounding the size of terms thanks to typing, we were able to bound the total number of messages which can be generated in a minimal witness of non-equivalence. This limitation seems to bring our process algebra closer to process algebra without terms, such as CCS, for which some (unbounded) equivalences such as bisimulation are decidable [48] and could constitute an interesting new approach to decidability if a tight bound on the number of terms can be computed.

A new decidable class: simple type-compliant acyclic protocols with nonces

Chapter 6 describes the first decidability result for trace equivalence for an unbounded number of sessions *with nonces*. It does so by defining a practical class of simple type-compliant acyclic protocols, which can be thought as simple acyclic tagged protocols. It introduces the notion of dependency graph to statically evaluate the high-level information flow of a type-compliant protocol based on its specification to define acyclic protocols. This notion was independently described in [47] to decide leakiness for tagged protocols. Relying on the typing result from Chapter 4, it supports only symmetric encryption but still deals with a large number of protocols from the literature, and is likely to be extended at the same time as the typing result. The notion of dependency graph is also prone to improvements: as it fundamentally describes an over-approximation in the causal dependencies between actions in the specification of a protocol, and introduces more edges than needed as a result, it can be refined, and pruned, by considering additional sources of information, such as already proved security properties to refine the abstraction. This would for instance be profitable when dealing with protocols using temporary secrets, which are currently considered cyclic. As secrecy properties can be expressed as equivalence properties, this result also provides a new decidable class of tagged protocols for these properties. Finally, Chapter 6 exposes an upper bound on the length of any minimal witness of non-equivalence (if existing) and thus could be useful to turn existing tools for bounded trace equivalence into tools able to check for full unbounded trace equivalence.

A new decidable class: ping-pong protocols without nonces

Chapter 7 also details one of the first class of protocols for which trace equivalence is decidable for an unbounded number of sessions. It focuses on ping-pong protocols without nonces and without pairing to provide a two-way reduction between cryptographic protocols and generalised deterministic pushdown automata, linking trace equivalence to the language equivalence of deterministic pushdown automata, proven to be decidable [63]. The reduction from automata to protocols moreover offers a very general undecidability result for trace inclusion and equivalence, underlying the inherent difficulty of finding interesting decidable classes. Looking at the complexity of the reductions between these models, it offers some insight on the complexity of trace equivalence for ping-pong protocols. The results from Chapter 7 also lead to a tool, `Cpp2dpa`, interfacing with the tool `lAlBlC` [51] for language equivalence checking, to automatically check for trace equivalence between such protocols. Improvements can be made to offer

more efficient verification. A more theoretical approach to extend the class of protocols we consider would be to move from traditional pushdown automata to automata with stacks of stacks or terms, for which deciding equivalence would likely be more difficult.

Existing decidability results for equivalence for an unbounded number of sessions, including those presented in this thesis, all rely on some determinism hypothesis, *e.g.* determinate or simple protocols, to be proven. These hypotheses also constrain the process algebra which is considered, for instance frequently barring else branches. Unfortunately, properties like untraceability rely on non-determinism in the protocol modelling to hold, as they are inherently related to the ability for protocols to conceal their actions. To be able to prove this property, as well as a number of other interesting properties for privacy, one would need to design new classes, and new proof methods, for the equivalence of non-deterministic protocols.

Despite the new decidability results offered by this thesis, their direct application to fully deployed cryptographic protocols still suffers from the high complexity cost of all the procedures designed so far. Even when accounting for the increase in raw computing performance of computers, scalability is a deep issue for protocol verification. Composition results [36] offer a possible approach, and, in general, considering modelling and proof frameworks with built-in composability seems like a fruitful direction.

In the process of designing and proving new classes of protocols for which equivalence is decidable, one crucial point lies in the choice of the restrictions to use. They should be practical enough to include as many realistic protocols and still provide a theoretical edge to prove decidability. Tagging schemes, and the generalisation we proposed with type-compliant protocols, along with typing results offer a great example of such a combination and lead to powerful results for both reachability and equivalence properties. To obtain more comprehensive decidability results, and possibly results effectively compatible with tool support, a promising approach would be to deepen our understanding of the structure of protocols and translate it into structures which enable powerful methods to be applied on them. Dependency graphs as described in Chapter 6 and [47] are probably a first step in that direction.

Bibliography

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th Symposium on Principles of Programming Languages (POPL'01)*. ACM Press, 2001.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1 – 70, 1999.

[3] M. Abadi and R. M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, 1996.

[4] R. Amadio and W. Charatonik. On name generation and set-based analysis in the Dolev-Yao model. In *13th Int. Conference on Concurrency Theory (CONCUR'02)*, 2002.

[5] M. Arapinis, T. Chothia, E. Ritter, and M. Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *23rd Computer Security Foundations Symposium (CSF'10)*, pages 107–121. IEEE Computer Society Press, 2010.

[6] M. Arapinis and M. Duflot. Bounding messages for free in security protocols. In *27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*, 2007.

[7] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008, Alexandria, VA, USA, October 27, 2008*, pages 1–10, 2008.

[8] D. Baelde, S. Delaune, and L. Hirschi. Partial order reduction for security protocols. In L. Aceto and D. de Frutos-Escrig, editors, *Proceedings of the 26th International Conference on Concurrency Theory (CONCUR'15)*, volume 42 of *Leibniz International Proceedings in Informatics*, pages 497–510, Madrid, Spain, Sept. 2015. Leibniz-Zentrum für Informatik.

[9] D. Baelde, A. Gacek, D. Miller, G. Nadathur, and A. Tiu. The bedwyr system for model checking over syntactic expressions. *CoRR*, abs/cs/0702116, 2007.

[10] D. Basin, J. Dreier, and R. Sasse. Automated symbolic proofs of observational equivalence. In *ACM CCS*, page ? ACM, 2015.

[11] M. Baudet. Deciding security of protocols against off-line guessing attacks. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 16–25, New York, NY, USA, 2005. ACM.

[12] M. Baudet. Deciding security of protocols against off-line guessing attacks. In *12th ACM Conference on Computer and Communications Security (CCS'05)*. ACM Press, 2005.

[13] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th Computer Security Foundations Workshop (CSFW'01)*. IEEE Computer Society Press, 2001.

[14] B. Blanchet, M. Abadi, and C. Fournet. Automated Verification of Selected Equivalences for Security Protocols. In *20th Symposium on Logic in Computer Science*, 2005.

[15] B. Blanchet and A. Podelski. Verification of cryptographic protocols: Tagging enforces termination. In *Foundations of Software Science and Computation Structures (FoSSaCS'03)*.

[16] S. Boehm and S. Goeller. Language equivalence of deterministic real-time one-counter automata is nl-complete. In *Proceedings of the 36th International Symposium on Mathematical Foundations of Computer Science (MFCS'11)*, volume 6907 of *LNCS*, pages 194–205, 2011.

[17] M. Boreale, R. D. Nicola, and R. Pugliese. Proof techniques for cryptographic processes. *SIAM Journal on Computing*, 31(3):947–986, 2001.

[18] M. Bruso, K. Chatzikokolakis, and J. den Hartog. Formal verification of privacy for RFID systems. In *23rd Computer Security Foundations Symposium (CSF'10)*, 2010.

[19] R. Chadha, Ş. Ciobâcă, and S. Kremer. Automated verification of equivalence properties of cryptographic protocols. In *21th European Symposium on Programming (ESOP'12)*, LNCS.

[20] V. Cheval. APTE: an algorithm for proving trace equivalence. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 587–592, 2014.

[21] V. Cheval and B. Blanchet. Proving more observational equivalences with proverif. In D. Basin and J. Mitchell, editors, *Proceedings of the 2nd International Conference on Principles of Security and Trust (POST'13)*, volume 7796 of *Lecture Notes in Computer Science*, pages 226–246, Roma, Italy, Mar. 2013. Springer Berlin Heidelberg.

[22] V. Cheval, H. Comon-Lundh, and S. Delaune. Trace equivalence decision: Negative tests and non-determinism. In *18th ACM Conference on Computer and Communications Security*.

[23] R. Chrétien, V. Cortier, and S. Delaune. From security protocols to pushdown automata. In *40th Int. Colloquium on Automata, Languages and Programming (ICALP'13)*, 2013.

[24] R. Chrétien, V. Cortier, and S. Delaune. Typing messages for free in security protocols: the case of equivalence properties. In *Proceedings of the 25th International Conference on Concurrency Theory (CONCUR'14)*, volume 8704 of *LNCS*, pages 372–386, Rome, Italy, Sept. 2014. Springer.

[25] R. Chrétien, V. Cortier, and S. Delaune. Checking trace equivalence: How to get rid of nonces? In P. Ryan and E. Weippl, editors, *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS'15)*, Lecture Notes in Computer Science, Vienna, Austria, Sept. 2015. Springer. To appear.

[26] R. Chrétien, V. Cortier, and S. Delaune. Decidability of trace equivalence for protocols with nonces. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF'15)*. IEEE Computer Society Press, June 2015. To appear.

[27] R. Chrétien, V. Cortier, and S. Delaune. From security protocols to pushdown automata. *ACM Transactions on Computational Logic*, 2015. To appear.

[28] R. Chrétien and S. Delaune. Formal analysis of privacy for routing protocols in mobile ad hoc networks. In D. Basin and J. Mitchell, editors, *Proceedings of the 2nd International Conference on Principles of Security and Trust (POST'13)*, volume 7796 of *Lecture Notes in Computer Science*, pages 1–20, Rome, Italy, Mar. 2013. Springer.

[29] S. Christensen, H. Huttel, and C. Stirling. Bisimulation equivalence is decidable for all context-free processes. *Information and Computation*, 121(2):143 – 148, 1995.

[30] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0, 1997.

[31] H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. In *14th International Conference on Rewriting Techniques and Applications (RTA'2003)*, volume 2706 of *LNCS*. Springer, 2003.

[32] H. Comon-Lundh and V. Cortier. Security properties: two agents are sufficient. In *Proc. of the 12th European Symposium On Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 99–113. Springer Verlag, April 2003.

[33] H. Comon-Lundh and V. Cortier. Computational soundness of observational equivalence. In *15th ACM Conference on Computer and Communications Security (CCS'08)*. ACM Press, 2008.

[34] H. Comon-Lundh, V. Cortier, and E. Zalinescu. Deciding security properties for cryptographic protocols. Application to key cycles. *ACM Transactions on Computational Logic (TOCL)*, 11(4), 2010.

[35] V. Cortier and S. Delaune. A method for proving observational equivalence. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*. IEEE Computer Society Press, 2009.

[36] V. Cortier and S. Delaune. Safely composing security protocols. *Formal Methods in System Design*, 34(1):1–36, Feb. 2009.

[37] V. Cortier and B. Smyth. Attacking and fixing Helios: An analysis of ballot secrecy. *Journal of Computer Security*, 21(1):89–148, 2013.

[38] A. Dallon. Reducing the number of agents in equivalence properties. Rapport de Master, 2015.

[39] S. Delaune, S. Kremer, and M. D. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, (4):435–487, July 2008.

[40] S. Delaune, M. D. Ryan, and B. Smyth. Automatic verification of privacy properties in the applied pi-calculus. In *IFIPTM'08: 2nd Joint iTrust and PST Conferences on Privacy, Trust Management and Security*, volume 263 of *International Federation for Information Processing (IFIP)*, pages 263–278. Springer, 2008.

[41] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communication of the ACM*, 24(8):533–536, 1981.

[42] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[43] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[44] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols*, Trento, Italia, 1999.

[45] S. Even and O. Goldreich. On the security of multi-party ping-pong protocols. In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 34–39, 1983.

[46] E. P. Friedman. The inclusion problem for simple languages. *Theor. Comput. Sci.*, 1(4):297–316, 1976.

[47] S. Fröschle. Leakiness is decidable for well-founded protocols? In *Proceedings of the 4th Conference on Principles of Security and Trust (POST'15)*, Lecture Notes in Computer Science, London, UK, Apr. 2015. Springer.

[48] J. F. Groote and H. Hüttel. Undecidable equivalences for basic process algebra. *Information and Computation*, 115:354–371, 1991.

[49] J. D. Guttman and F. J. Thayer. Protocol independence through disjoint encryption. In *13th Computer Security Foundations Workshop (CSFW'00)*. IEEE Comp. Soc. Press, 2000.

[50] P. Henry and G. Sénizergues. Lalblc a program testing the equivalence of dpda's. In *18th International Conference on Implementation and Application of Automata (CIAA 2013)*, volume 7982 of *Lecture Notes in Computer Science*, pages 169–180, Halifax, NS, Canada, 2013. Springer.

[51] P. Henry and G. Sénizergues. Lalblc a program testing the equivalence of dpda's. In *CIAA*, pages 169–180, 2013.

[52] ICAO. Machine readable travel document. Technical Report 9303, International Civil Aviation Organization, 2008.

[53] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 147–166. Springer-Verlag, march 1996.

[54] G. Lowe. Towards a completeness result for model checking of security protocols. In *Proc. of the 11th Computer Security Foundations Workshop (CSFW'98)*. IEEE Computer Society Press, 1998.

[55] J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, CCS '01, pages 166–175, New York, NY, USA, 2001. ACM.

[56] J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *8th ACM Conference on Computer and Communications Security*, 2001.

[57] D. L. Mitchell, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. 1999.

[58] R. Needham and M. Schroeder. Using encryption for authentification in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[59] D. J. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[60] R. Ramanujam and S. P. Suresh. Tagging makes secrecy decidable with unbounded nonces as well. In *23rd Conference of Foundations of Software Technology and Theoretical Computer Science (FSTTCS'03)*, LNCS, pages 363–374. Springer, 2003.

[61] S. Santiago, S. Escobar, C. Meadows, and J. Meseguer. A formal definition of protocol indistinguishability and its verification using Maude-NPA. In *Security and Trust Management - 10th International Workshop, STM 2014, Wroclaw, Poland, September 10-11, 2014. Proceedings*, pages 162–177, 2014.

[62] G. Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *24th Int. Coll. on Automata, Languages and Programming (ICALP'97)*, LNCS, 1997.

[63] G. Sénizergues. L(A)=L(B)? Decidability results from complete formal systems. *Theor. Comput. Sci.*, 251(1-2):1–166, 2001.

[64] C. Stirling. Deciding DPDA equivalence is primitive recursive. In *29th International Colloquium on Automata, Languages and Programming ICALP'02*, LNCS. Springer, 2002.

[65] A. Tiu and J. E. Dawson. Automating open bisimulation checking for the spi calculus. In *23rd IEEE Computer Security Foundations Symposium (CSF'10)*, pages 307–321, 2010.

[66] A. Tiu, R. Goré, and J. E. Dawson. A proof theoretic analysis of intruder theories. *Logical Methods in Computer Science*, 6(3), 2010.

Appendix A

Well-typed executions

Section A.1 will focus on the proof of completeness of the said procedure, as stated in Proposition 4.2.3 later in the appendix. Section A.2 will provide the complete proofs of Proposition 4.1.1 and Theorem 4.1.1.

A.1 Proof of Proposition 4.2.3

The proof of Proposition 4.2.3 requires a number of technicalities so as to reduce a concrete witness of non-equivalence between two protocols into a valid output of the algorithm described in Section 4.2. In particular, recipes used by the attacker to discriminate between two frames need to be modified to be proper tests in the symbolic frames introduced by the said algorithm. Section A.1.1 will deal with this aspect; while Section A.1.2 will define a more operational notion of static equivalence and formally link symbolic traces from the algorithm to concrete executions of the protocols, which will be needed to finally prove Proposition 4.2.3.

A.1.1 Simplifying recipes

In this section, we present how equalities between arbitrary recipes can be transformed into a set of equalities between recipes sharing interesting properties, defined in the next definitions. In the following, ϕ and ψ represent two (concrete) frames, while ϕ_S and ψ_S are two symbolic frames such that $\phi = \phi_S\lambda_P$ and $\psi = \psi_S\lambda_Q$, where $\lambda_P = (\theta\phi)\downarrow$ and $\lambda_Q = (\theta\psi)\downarrow$ and θ is a substitution such that $(vars(\phi_S) \cup vars(\psi_S)) \subseteq dom(\theta)$ and $img(\theta) \subseteq \mathcal{T}_0(\Sigma, \Sigma_0 \cup \mathcal{W})$. These relations are justified later by Lemmas A.1.14 and A.1.16 in Section A.1.2.

The notions of precompact and compact recipes restrict the tests that can be made by the attacker when trying to distinguish between two frames. Lemma A.1.13 in Section A.1.2 will prove later this is not, in our setting, an actual restriction.

Definition A.1.1 (precompact recipe)**.** Given a frame ϕ, a recipe R is said to be *ϕ-precompact* if:

- $R\phi\downarrow$ is a message
- R contains only destructors
- $R\phi\downarrow$ is neither a pair nor an encryption by a key deducible in ϕ

We now introduce the notion of *symbolic* second-order trace, which is helpful to reason on the objects generated in the decision algorithm of Section 4.2.

Definition A.1.2. (tr_S, ϕ_S) is a *symbolic* second-order trace of a protocol P if there exists a bijective renaming ρ from $vars(\mathsf{tr}_S\phi_S\downarrow)$ such that $(\mathsf{tr}_S\rho, \phi_S\rho) \in \mathsf{trace}(P)$. In that case, ϕ_S is called a *symbolic* frame.

Definition A.1.3 (compact recipe)**.** Given a symbolic frame ϕ_S, R is said to be ϕ_S-*compact* if R is ϕ_S-precompact and $R\phi_S\downarrow$ is not a variable.

A recipe R is said to be destructor-only if $R \in \mathcal{T}(\Sigma_{\mathsf{d}}, \Sigma_0 \cup \mathcal{W} \cup \mathcal{X})$, *i.e.* contains no constructor.

We introduce a new predicate on recipe, msg, with the natural semantics: $\phi \models \mathsf{msg}(R)$ if $R\phi\downarrow$ is a message (with or without variables, *i.e.* an element of $\mathcal{T}_0(\Sigma_c, \Sigma_0 \cup \mathcal{N} \cup \mathcal{X})$). Given a term, we introduce a rightmost-first order on its positions, which corresponds to the anti-lexicographic order on positions in a term, denoted by $<$. $\ll$ denotes the order on positions such that $p \ll q$ iff q is a strict prefix of p. Note that: $p \ll q \Rightarrow p < q$. We will sometimes refer to a term as the rightmost term verifying a property, *i.e.* the term verifying this property whose position is the lowest by the $<$ order.

Definition A.1.4 (transformation of concrete recipes for ϕ)**.** Given ϕ, a destructor-only recipe R, ϕ_S and θ as introduced earlier, we define this transformation $\mathcal{T}$ as follows:

- if there exists $R' \in st(R)$ such that $R = C[R']$ and R' is the rightmost recipe verifying one the following two conditions:
  1. if $R' = \mathsf{dec}(R_1, R_2)$ and there exists a variable x such that $R_1\phi_S\downarrow = \mathsf{enc}(t, x)$ and $R_2 \neq x$ for some term t, then $\mathcal{T}(R, \mathcal{R}) = (C[\mathsf{dec}(R_1, x)], \mathcal{R} \cup \{\mathsf{msg}(\mathsf{dec}(R_1, x))\})$
  2. else, and if there exists a variable y such that $R'\phi_S\downarrow = y$ and $R' \neq y$, then $\mathcal{T}(R, \mathcal{R}) = (C[y\theta]\Downarrow, \mathcal{R} \cup \{R' = y\})$
- if no such recipe exists, $\mathcal{T}(R, \mathcal{R}) = (R, \mathcal{R})$;

where $\Downarrow$ is the normal form associated to the rewriting rules $\pi_i(\langle x_1, x_2\rangle) \to x_i$ and $\mathsf{dec}(\mathsf{enc}(x, y), z) \to x$.

We denote the iterated application of $\mathcal{T}$ to $(R, \emptyset)$ by $\mathcal{T}^*(R)$ or $\mathcal{T}^m(R)$ (when iterated m times).

The iterated transformation $\mathcal{T}^*$ aims at transforming a ϕ-precompact recipe (which is still a quite general class of recipes) into a ϕ_S-compact recipe, *i.e.* a recipe which can reduce properly in a symbolic frame and will satisfy somewhat similar equalities. The next lemmas will gradually prove the properties we need for $\mathcal{T}^*$, ultimately ending with Lemmas A.1.10, A.1.11 and A.1.12.

Lemma A.1.1 (consistency of $\mathcal{R}$)**.** Let R be a destructor-only recipe, $\mathcal{T}^*(R) = (R^*, \mathcal{R})$: if $R\phi\downarrow$ is a message, $\phi_S \models \mathcal{R}$.

Proof. A test $\mathsf{msg}(\mathsf{dec}(R, x))$ or $R = x$ is added to $\mathcal{R}$ in Definition A.1.4 only if $R\phi_S\downarrow = \mathsf{enc}(t, x)$ where t is a term (in the first case) or if $R\phi_S\downarrow = x$ (in the second case). As ϕ_S only contains symbolic messages in its image; t has to be a symbolic message. □

This lemma witnesses the fact that the equalities we insert in $\mathcal{R}$, which correspond to equalities holding in the concrete frame, actually hold in its symbolic version. Lemmas A.1.2 and A.1.3 witness rather general properties of destructor-only recipes and of the $\Downarrow$ reduction.

Lemma A.1.2. If R is a recipe such that $R\phi\downarrow$ is a message, then $R\Downarrow\phi\downarrow = R\phi\downarrow$

Proof. If $R\phi\downarrow$ is a message, every reduction step of the form $\mathsf{dec}(\mathsf{enc}(x, y), z) \Rightarrow x$ in an innermost derivation in $R\Downarrow$ happens with $y\phi = z\phi$ and is then actually a reduction step of the form $\mathsf{dec}(\mathsf{enc}(x, y), y) \to x$ in $R\phi$. □

Lemma A.1.3. If R is a destructor-only recipe and ϕ a frame such that $R\phi\downarrow$ is a message, then for every subterm R' of R, $R'\phi\downarrow$ is a message.

Proof. Suppose there exists a highest (in terms of position, *i.e.* closest to the root) subterm R' of R such that $R'\phi\downarrow$ is not a message and $R = C[R']$ (and C is linear). Let us proceed by induction on C to show that $C[R']\phi\downarrow$ is not a message either. Note that, because R is destructor-only, we only consider C to be destructor-only.

- $C = \_$: then $R = R'$ and $R\phi\downarrow$ is not a message.
- $C = \mathsf{proj}_i(C')$ and $C'[R']\phi\downarrow$ is not a message: then $\mathsf{proj}_i(C'[R'])\phi\downarrow$ will not be a message either.
- $C = \mathsf{dec}(R'', C')$, $C'[R']\phi\downarrow$ and $C'[R']\phi\downarrow$ is not a message. For $C[R']\phi\downarrow$ to be a message, it would require $R''\phi\downarrow = \mathsf{enc}(s, C'[R']\phi\downarrow)$ for some message s. As ϕ only contains messages in its image and $\mathsf{enc}(s, C'[R']\phi\downarrow)$ is not one ($C'[R']\phi\downarrow$ is not a message) the enc function symbol would need to appear in C, which is destructor-only: contradiction. Hence $C[R']\phi\downarrow$ is not a message.
- $C = \mathsf{dec}(C', R'')$, $C'[R']\phi\downarrow$ is not a message: $C[R']\phi\downarrow$ being a message would imply $C'[R']\phi\downarrow = \mathsf{enc}(s, R''\phi\downarrow)$ for some term s. As before, because ϕ contains only messages in its image and $C'[R']\phi\downarrow$ is not, the enc symbol need to occur in $C'[R']$ which is destructor-only, as a subterm of R.

Thus every subterm R' of R is such that $R'\phi\downarrow$ is a message. □

Lemma A.1.4 (preservation of normal forms)**.** If R is a destructor-only recipe, $(R\theta)\phi\downarrow$ is a message, $(\bar{R}, \mathcal{R}) = \mathcal{T}(R)$, then $(R\theta)\phi\downarrow = (\bar{R}\theta)\phi\downarrow$.

Proof. Using the notations introduced in Definition A.1.4, $R = C[R']$, R' is the rightmost subterm of R verifying one of the two conditions of $\mathcal{T}$.

1. if $R' = \mathsf{dec}(R_1, R_2)$, $R_1\phi_S\downarrow = \mathsf{enc}(t, x)$ and $R_2 \neq x$: $\bar{R} = C[\mathsf{dec}(R_1, x)]$. We have that $\bar{R}\theta = C[\mathsf{dec}(R_1\theta, x\theta)]\theta$. And thus $(\bar{R}\theta)\phi\downarrow = (C[\mathsf{dec}(R_1\theta, x\theta)]\theta)\phi\downarrow = (C[t\theta]\theta)\phi\downarrow$. As $R_1\phi_S\downarrow = \mathsf{enc}(t, x)$ and $(R\theta)\phi\downarrow$ is a message, $(R'\theta)\phi\downarrow$ is a message too (Lemma A.1.3) and $(R'\theta)\phi\downarrow = (R_1\theta\phi)\downarrow = t\theta\phi$. Hence $(C[t\theta]\theta)\phi\downarrow = (C[R']\theta)\phi\downarrow = (R\theta)\phi\downarrow$, giving the result.
2. else, and if $R'\phi_S\downarrow = y$ and $R' \neq y$: $\bar{R} = C[y\theta]\Downarrow$. We have $y\theta\phi\downarrow = (R'\theta)\phi\downarrow$. Indeed, as $R'\phi_S\downarrow = y$, $R'\phi_S\downarrow\theta\phi\downarrow = y\theta\phi\downarrow$. Then $(R'\theta)\phi\downarrow = y\theta\phi\downarrow$, as $\phi = \phi_S\lambda_P$ (works with $\psi = \psi_S\lambda_Q$ too, thanks to Lemma A.1.16). To conclude, we show that $(R\theta)\phi\downarrow = (C[(R'\theta)\phi\downarrow]\theta)\phi\downarrow$, and as $(R\theta)\phi\downarrow$ is a message and by Lemma sA.1.2, this is equal to $(C[y\theta\phi\downarrow]\theta)\phi\downarrow = (C[y\theta]\theta)\phi\downarrow = (C[y\theta]\Downarrow\theta)\phi\downarrow = (\bar{R}\theta)\phi\downarrow$.

□

This lemma proves that the transformation $\mathcal{T}$, from the point of view of the concrete frame, does not alter the normal form of tests (up to a particular substitution θ). Previous lemmas are stated with ϕ and ϕ_S, but can be symmetrically applied with ψ and ψ_S. When the context is not obvious, we will denote by $\mathcal{T}_\phi$ the transformation introduced at Definition A.1.4 when applied with ϕ_S, and $\mathcal{T}_\psi$ when applied with ψ_S.

Lemma A.1.5 (termination of $\mathcal{T}^*$)**.** $\mathcal{T}$ is deterministic and $\mathcal{T}^*(R)$ is well-defined.

Proof. Introduce an ordering on variables based on how soon they appear in the trace (tr_S, ϕ_S) and consider the induced multi-set order $<_{\mathsf{var}}$. The second item in Definition A.1.4 strictly reduces this measure. Now consider the number of destructor of a (destructor-only) recipe R, plus the number of variables in $dom(\phi)$ (without counting the variables in $dom(\theta)$). Let $<_{\mathsf{size}}$ be the order induced by this measure. The first item in Definition A.1.4 strictly reduces this measure. Finally, let $<$ be the lexicographical order built on $(<_{\mathsf{var}}, <_{\mathsf{size}})$. The transformation $\mathcal{T}$ decreases its induced measure. □

Lemmas A.1.6, A.1.7 and Corollary A.1.1 provide the general invariants for the transformation $\mathcal{T}$: mostly that it operates locally, does not introduce new constructors and preserves the fact for a subterm of being a message.

Lemma A.1.6. If R is ϕ-precompact, then for every $n \in \mathbb{N}$, $\mathcal{T}_\phi^n(R)$ is destructor-only.

Proof. Suppose there exists $n_0 \in \mathbb{N}$ such that $R_{n_0} = \mathcal{T}^{n_0}(R)$ contains a constructor c at position p. Let us further assume p is the highest position where a constructor occurs.

- if $p = \epsilon$: as $(R_{n_0}\theta)\phi\downarrow = R\phi\downarrow$ by Lemma A.1.4, R cannot be ϕ-precompact,
- if $p > \epsilon$ and $c = \langle \_, \_ \rangle$: c occurs below a destructor d (as p is the highest position a constructor can appear).
  - If $d = \mathsf{proj}_i$ for $i \in \{1, 2\}$: as R is destructor-only, c is introduced by a replacement $C[y\theta]\Downarrow$ and because $\mathsf{proj}_i(\langle x_1, x_2 \rangle) \Rightarrow x_i$, c would have been reduced.
  - If $d = \mathsf{dec}$ and c is in plaintext position of d: if c is not reduced, $(R_{n_0}\theta)\phi\downarrow$ is not a message, then $(R_{n_0}\theta)\phi\downarrow = R\phi\downarrow$, by Lemma A.1.4, implies $R\phi\downarrow$ is not a message either, and thus not ϕ-precompact.
  - Else, if $d = \mathsf{dec}$ and c is in key position of d: the key is not atomic, and similarly $R\phi\downarrow$ is not a message, and thus not ϕ-precompact.
- if $p > \epsilon$ and $c = \mathsf{enc}(\_, \_)$: the same reasoning as the previous case can be applied (interverting the first two subcases).

Hence R_{n_0} cannot contain constructors if R is ϕ-precompact. □

Lemma A.1.7. Let R^{init} be a destructor-only recipe and $R = \mathcal{T}_\phi^n(R^{\text{init}})$. If $R = C[R']_p$, R is destructor-only, $(R'\theta)\phi\downarrow$ is a message and p is lesser w.r.t. $<$ than the next position where $\mathcal{T}_\phi$ is applied on R, then $R|_q\phi_S\downarrow$ is a message for any $q \ll p$ or $q = p$.

Proof. We prove by induction that $R|_q\phi_S\downarrow$ is a message.

- if $R|_q = \mathsf{w}$, $\mathsf{w}\phi_S$ is a message,
- if $R|_q = x$, $x\phi_S = x$ is a message,
- if $R|_q = \mathsf{enc}(R_1, R_2)$ or $R|_q = \langle R_1, R_2 \rangle$, $R_1\phi_S\downarrow$ and $R_2\phi_S\downarrow$ are messages by induction hypothesis, then $R|_q\phi_S\downarrow$ is a message too,
- if $R|_q = \mathsf{proj}_i(R'')$ and $R''\phi_S\downarrow$ is a message:
  - if $R''\phi_S\downarrow = x$ for some variable x: a variable in R'' can only be introduced in key position of a dec, not a proj_i, impossible.
  - if $R''\phi_S\downarrow = \langle R_1, R_2 \rangle$: then $\mathsf{proj}_i(\langle R_1, R_2 \rangle)\phi_S\downarrow$ is a message,
  - if $R''\phi_S\downarrow = \mathsf{enc}(u, v)$: as $(R''\theta)(\phi_S\lambda_P)\downarrow = (R''\theta)\phi\downarrow$, there exist u', v' two terms such that $(R''\theta)\phi\downarrow = \mathsf{enc}(u', v')$. Thus $(\mathsf{proj}_i(R'')\theta)\phi\downarrow$ is not a message, and $(R'\theta)\phi\downarrow$ is not a message either, by Lemma A.1.3 as R is destructor-only. Contradiction.
- if $R|_q = \mathsf{dec}(R_1, R_2)$, $R_1\phi_S\downarrow$ and $R_2\phi_S\downarrow$ are messages:
  - if $R_1\phi_S\downarrow = x$ for some variable x: a variable can only occur in a key position, impossible. Either $R_1 = x$, which can only occur in a key position, impossible; or $R_1 \neq x$ and $R_1\phi_S\downarrow = x$, in which case $\mathcal{T}$ would be applied at position $q.1 < p$
  - if $R_1\phi_S\downarrow = \langle u, v \rangle$: see the third point of the previous case.
  - if $R_1\phi_S\downarrow = \mathsf{enc}(u, x)$ for some term u and some variable x. By Definition A.1.4, as $\mathcal{T}$ was applied at position q, $R_2 = x$ and $\mathsf{dec}(R_1, x)\phi_S\downarrow$ is a message.
  - if $R_1\phi_S\downarrow = \mathsf{enc}(u, k)$ for some term u and some non-variable atom k: then $R_2\phi_S\downarrow \neq x$ for any variable x, as R_2 is a position lesser than p. Thus $R_2\phi_S\downarrow = k'$ for some atom k', and $R_2\phi\downarrow = k'$ as k' is not a variable. Similarly $R_1\phi\downarrow = \mathsf{enc}(u', k)$ for some term u'. $(R\theta)\phi\downarrow$ is a message implies (as R is destructor-only and by Lemma A.1.3) $k = k'$ and that $\mathsf{dec}(R_1, R_2)\phi_S\downarrow$ is a message too.

□

Corollary A.1.1. Let R^{init} be a destructor-only recipe and $R = \mathcal{T}_\phi^n(R^{\text{init}})$. If $R = C[R']_p$, R is destructor-only, $(R'\theta)\phi\!\downarrow$ is a message and $\mathcal{T}_\phi(R) = (R, \mathcal{R})$ then $R|_q\phi_S\!\downarrow$ is a message for any $q \ll p$ or $q = p$.

Proof. Same proof as Lemma A.1.7, except we invoke the fact the transformation does not alter R any more in cases where we derive an impossibility. □

The following lemma intends to show that equalities between transformed test $\mathcal{T}^*(R)$ actually correspond to the unification performed by the algorithm at step 2. Indeed, the normal forms of ϕ_S-compact recipes are encrypted subterms that can be unified in this step.

Lemma A.1.8 (compacification effect of $\mathcal{T}^*$)**.** With previous notations, if $\mathcal{T}^*(R) = (R^*, \mathcal{R})$: if R is ϕ-precompact then R^* is ϕ_S-compact.

Proof. As R is ϕ-precompact, by Lemma A.1.6, any (iterated) application of $\mathcal{T}_\phi$ to R yields a destructor-only recipe. We then use Lemma A.1.7 repeatedly and finally Corollary A.1.1 with $p = \epsilon$ (the root position) and for $q = p$ to conclude $R^*\phi_S\!\downarrow$ is a message. Invoking Lemma A.1.4 at each step finally gets us $R\phi\!\downarrow = (R^*\theta)\phi\!\downarrow$. As $(R^*\theta)\phi\!\downarrow = R^*\phi_S\!\downarrow\lambda_P$, we can conclude that $R^*\phi_S\!\downarrow$ is not a pair nor an encryption with a deducible key (for the latter, consider the case where $R^*\phi_S\!\downarrow = \mathsf{enc}(u_1, u_2)$: as R is ϕ-precompact, $u_2\lambda_P$ is not deducible in ϕ; but if u_2 were deducible in ϕ_S, $u_2\lambda_P$ would be in $\phi \cup \lambda_P$, and thus in ϕ. Contradiction). □

Lemma A.1.9 (uniformity of $\mathcal{T}$)**.** Let $\phi, \phi_S, \psi, \psi_S$ be as introduced, R be a destructor-only recipe, $\mathcal{T}_\phi^*(R) = (R_1^*, \mathcal{R}_1)$ and $\mathcal{T}_\psi^*(R) = (R_2^*, \mathcal{R}_2)$. If $\psi_S \models \mathcal{R}_1$ and $\phi_S \models \mathcal{R}_2$, then $R_1^* = R_2^*$.

Proof. Let $(R_1^k, \mathcal{R}_1^k) = \mathcal{T}_\phi^k(R)$ and $(R_2^k, \mathcal{R}_2^k) = \mathcal{T}_\psi^k(R)$ be the iterated application of $\mathcal{T}$ with both symbolic frames. We will prove inductively that $R_1^k = R_2^k$ and $\mathcal{R}_1^k = \mathcal{R}_2^k$.

- for $k = 0$, the initial recipe, R, is identical in both cases and $\mathcal{R}_1^0 = \emptyset = \mathcal{R}_2^0$.
- Let us assume we obtained the result up to some k: $R_1^k = R_2^k = R^k$. Note that $\mathcal{R}_i^k \subseteq \mathcal{R}_i$ for $i \in \{1, 2\}$ implies $\phi_S \models \mathcal{R}_2^k$ and $\psi_S \models \mathcal{R}_1^k$. Suppose now that $R^k = C[R']|_p$ and p is the lowest position w.r.t. $<$ such that any of the two rules in Definition A.1.4 can apply with *either* ϕ_S or ψ_S. For instance, suppose it is true for ϕ_S:

  1. if $R' = \mathsf{dec}(R_1, R_2)$, there exist a variable x and a term t such that $R_1\phi_S\!\downarrow = \mathsf{enc}(t, x)$, $R_2 \neq x$, then $R_1^{k+1} = C[\mathsf{dec}(R_1, x)]_p$ and $\mathcal{R}_1^{k+1} = \mathcal{R}^k \cup \{\mathsf{msg}(\mathsf{dec}(R_1, x))\}$. As $\psi_S \models \mathcal{R}_1$ and $\mathcal{R}_1^{k+1} \subseteq \mathcal{R}_1$, $\psi_S \models \mathsf{msg}(\mathsf{dec}(R_1, x))$. Hence $R_1\psi_S\!\downarrow = \mathsf{enc}(s, x)$ for some term s and the same rule of $\mathcal{T}_\psi$ can be applied at the same position, and will, as p is the lowest position where a rule of $\mathcal{T}$ is applicable for both symbolic frames, and then: $R_2^{k+1} = R_1^{k+1}$ and $\mathcal{R}_2^{k+1} = \mathcal{R}_1^{k+1}$.
  2. If there exists a variable y such that $R'\phi_S\!\downarrow = y$ and $R' \neq y$, then $R_1^{k+1} = C[y\theta]\!\Downarrow$ and $\mathcal{R}_1^{k+1} = \mathcal{R}^k \cup \{R' = y\}$. As $\psi_S \models \mathcal{R}_1$ and $\mathcal{R}_1^{k+1} \subseteq \mathcal{R}_1$, $\psi_S \models R' = y$, *i.e.* $R'\psi_S\!\downarrow = y$. The same rule of $\mathcal{T}_\psi$ can thus be applied at the same position, and will, as p is the lowest position where a rule of $\mathcal{T}$ is applicable for both symbolic frames, and then: $R_2^{k+1} = R_1^{k+1}$ and $\mathcal{R}_2^{k+1} = \mathcal{R}_1^{k+1}$.

  The case where p corresponds to a position *w.r.t.* ψ_S is handled symmetrically.

Applying that result for n such that $\mathcal{T}_\phi^*(R) = \mathcal{T}_\phi^n(R)$ leads to the final result. □

The next two lemmas finally ensure $\mathcal{T}^*$ does not alter the equalities in any nefarious way. Together they demonstrate that if a test holds in a concrete concrete frame, the transformed equality, up to a unification performed at step 2 of the algorithm, will hold in the symbolic frame.

Lemma A.1.10 (soundness of $\mathcal{T}^*$)**.** Let R_1, R_2 be two destructor-only recipes, ψ, ψ_S, λ_Q be as expected, $(R_1^*, \mathcal{R}_1) = \mathcal{T}_\psi^*(R_1)$ and $(R_2^*, \mathcal{R}_2) = \mathcal{T}_\psi^*(R_2)$. Then, for any $i \in \{1, 2\}$, $R_i\psi{\downarrow} = R_i^*\psi_S\lambda_Q{\downarrow}$; and thus $R_1^*\psi_S{\downarrow} = R_2^*\psi_S{\downarrow}$ implies $R_1\psi{\downarrow} = R_2\psi{\downarrow}$.

Proof. Iteration of Lemma A.1.4 gives $R_i\psi{\downarrow} = (R_i^*\theta)\psi{\downarrow}$; and we assumed $\psi = \psi_S\lambda_Q$ with $\lambda_Q = (\theta\psi){\downarrow}$. Hence, $R_i^*\psi_S\lambda_Q{\downarrow} = (R_i^*\theta)\psi_S(\theta\psi){\downarrow} = (R_i^*\theta)\psi{\downarrow} = R_i\psi{\downarrow}$. □

Lemma A.1.11 (completeness of $\mathcal{T}^*$)**.** Let R_1, R_2 be two ϕ-precompact recipes, ϕ, ϕ_S, λ_P be as expected, $(R_1^*, \mathcal{R}_1) = \mathcal{T}_\phi^*(R_1)$ and $(R_2^*, \mathcal{R}_2) = \mathcal{T}_\phi^*(R_2)$. Then: $R_1\phi{\downarrow} = R_2\phi{\downarrow}$ implies there exists σ which is a mgu of two ϕ_S-compact recipes in ϕ_S such that $R_1^*\phi_S\sigma{\downarrow} = R_2^*\phi_S\sigma{\downarrow}$.

Proof. Iteration of Lemma A.1.4 gives $R_i\phi{\downarrow} = R_i^*\phi{\downarrow}$; and we assumed that $\phi = \phi_S\lambda_P$. Then, $R_1\phi{\downarrow} = R_2\phi{\downarrow}$ is the same as $R_1^*\phi_S\lambda_P{\downarrow} = R_2^*\phi_S\lambda_P{\downarrow}$. Thus $\sigma = mgu(R_1^*\phi_S, R_2^*\phi_S) \neq \bot$. As R_1^* and R_2^* are ϕ_S-compact (Lemma A.1.8), we are done. □

Unfortunately, the transformation $\mathcal{T}$ may, in some cases, transform a recipe reducing to a term which was not a message into a new recipe reducing to a symbolic message. The next lemma ensures these special case can be handled in the main proof of completeness of the algorithm.

Lemma A.1.12 (preservation of messages)**.** Let ϕ, ϕ_S, ψ, ψ_S be as expected, R a ϕ-precompact recipe such that $\mathcal{T}_\phi^*(R) = (R^*, \mathcal{R}_1)$, $\psi_S \models \mathcal{R}_1$, $\mathcal{T}_\psi^*(R) = (R^*, \mathcal{R}_2)$ and $\phi_S \models \mathcal{R}_2$. If $R\phi{\downarrow}$, $R^*\phi_S{\downarrow}$, $R^*\psi_S{\downarrow}$ are messages but $R\psi{\downarrow}$ is not, then there either exist R_1^0 and R_2^0 two ϕ-precompact recipes such that $R_1^0\phi{\downarrow} = R_2^0\phi{\downarrow}$ and $R_1^0\psi{\downarrow} \neq R_2^0\psi{\downarrow}$ or there exists a ϕ_S-compact recipe R_0 such that $R_0\phi_S{\downarrow}$ is a message but $R_0\psi_S{\downarrow}$ is not.

Proof. Let p be the minimal position in R of a destructor d which is not reduced in $R\psi{\downarrow}$. Suppose $\mathcal{T}^*(R) = \mathcal{T}^n(R)$ (note that Lemma A.1.9 guarantees that $\mathcal{T}_\psi = \mathcal{T}_\phi$ when applied on R). Let R_i be a shortcut for the first argument of $\mathcal{T}^i(R)$. Let $q_1, q_2, \ldots, q_n$ be the successive positions where $\mathcal{T}$ is applied; and $i_0 \in \{1, \ldots, n\}$ such that $\forall 0 \leq j < i_0$, $(R_j\theta)\psi{\downarrow}$ contains d at position p and $(R_{i_0}\theta)\psi{\downarrow}$ does not. Let $\mathcal{P}$ be the set of positions in R. We define a partition of $\mathcal{P}$ as follows:

$$\mathcal{P} = \{p\} \cup \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \mathcal{P}_4 \cup \mathcal{P}_5$$

where:

- $\mathcal{P}_1$ is the set of positions on the "plaintext line" of p (always-leftmost children of p),
- $\mathcal{P}_2$ is the set of positions q such that $q < p$ and $q \notin \mathcal{P}_1$,
- $\mathcal{P}_3$ is the set of strict parents of p,
- $\mathcal{P}_4$ is the set of always-leftmost children of elements of $\mathcal{P}_3$ which are not in $\mathcal{P}_3 \cup \mathcal{P}_1 \cup \{p\}$,
- $\mathcal{P}_5$ is the set of positions greater than p w.r.t. $<$ which are not in $\mathcal{P}_3$ nor $\mathcal{P}_4$.

The core argument of this proof relies on a disjunction on the nature of q_{i_0}:

- if $q_{i_0} \in \mathcal{P}_2 \cup \mathcal{P}_5$: impossible, as any transformation by $\mathcal{T}$ at these positions cannot affect anything at position p. Indeed all these elements appears below a key position, *i.e.* a position which is the key of some dec.
- $q_{i_0} = p$: impossible, as if $\mathcal{T}$ were applied at this position, it would require $R_{i_0-1}|_p\psi_S{\downarrow}$ to be some variable x, and thus $(R_{i_0-1}\theta)\psi{\downarrow}$ would not contain d (as $\psi = \psi_S\lambda_Q$).
- $q_{i_0} \in \mathcal{P}_1$: by definition of i_0, $(R_{i_0-1}\theta)\psi{\downarrow}$ contains d at position p while $(R_{i_0}\theta)\psi{\downarrow}$ does not and $q_{i_0} < p$ (by definition of $\mathcal{P}_1$). Let $R_{i_0-1} = C[R']_{q_{i_0}}$. Because $\mathcal{T}$ is applied at q_{i_0} and d deleted at this step, necessarily $R'\psi_S{\downarrow} = x$ for some variable x and $R_{i_0} = C[x\theta]_{q_{i_0}}{\Downarrow}$; and d is deleted through the $\Downarrow$-reduction. Two cases can occur *a priori*:

– $d = \mathsf{proj}_i$. In that case, the same reductions appear with ϕ: hence $x\theta$ has to contain the pair which is deleted by d, and as recipes are the same no matter the frame we consider, the same reduction could occur in ψ, hence d would not occur in $R\psi\downarrow$ (We use Lemma A.1.4 to get the preservation of normal forms through the process). Impossibility.

– $d = \mathsf{dec}$. Then there exists a recipe in the key position of d at position $p.2 < p$: $R_2'^0$. Because $(R_2'^0\theta)\psi\downarrow$ is a message (minimality of p) and Lemma A.1.7 (third item, $p.2 < q_{i_0}$), $R_2'^0\psi_S\downarrow$ is a message. As R is ϕ-precompact and by Lemma A.1.6, $R_2'^0$ is destructor-only. Note that $R_2'^0$ may contain variables in key positions, due to the first item of Definition A.1.4. Because keys are atomic, for each variable in key position x, $x\lambda_P$ and $x\lambda_Q$ are atomic, thus there exist destructor-only recipe R_x for each of them. So we define $R_2^0 = R_2'^0[R_x/x]$. R_2^0 is now ϕ-precompact, $(R_2'^0\theta)\psi\downarrow = R_2^0\downarrow\psi$ and $(R_2'^0\theta)\phi\downarrow = R_2^0\downarrow\phi$. As $\Downarrow$-reduction only occurs at recipe level, there also exists a recipe R_1^0 for the encryption key used by the constructor reduced by d. Because keys are atomic, we also can always assume R_1^0 to be ϕ-precompact in this setting. We now show that $R_1^0\psi\downarrow \neq R_2^0\psi\downarrow$ and $R_1^0\phi\downarrow = R_2^0\phi\downarrow$. These equalities are derived from the fact that $R\psi\downarrow$ still contains d but $R\phi\downarrow$ is a message. Hence we obtain two recipes as in the statement of the lemma.

- $q_{i_0} \in \mathcal{P}_3$: then $R_{i_0-1}|_{q_{i_0}.1}\psi_S\downarrow = \mathsf{enc}(t, x)$ for some term t and variable x, and $p \ll q_{i_0}.2$ (or else $R_{i_0-1}|_{q_{i_0}.1}\psi_S\downarrow$ would not reduce if it included d). We get that $R_{i_0} = C[\mathsf{dec}(R_{i_0-1}|_{q_{i_0}.1}, x)]_{q_{i_0}}$. Let $R_0 = R_{i_0-1}|_{q_{i_0}.2}$. In particular, p is a position of R_0. As $q_{i_0}.2 < q_{i_0}$, R_0 is ϕ_S-compact (Lemmas A.1.6 and A.1.7). We now want to show that $R_0\psi_S\downarrow$ is not a message: as $(R_0\theta)\psi\downarrow$ contains d by definition, if $R_0\psi_S\downarrow$ did not, $R_0\psi_S\downarrow\lambda_Q\downarrow = (R_0\theta)\psi\downarrow$ would not either, which contradicts our hypothesis. Hence we get our recipe R_0 as in the second case of the statement of the lemma.

- $q_{i_0} \in \mathcal{P}_4$: we apply the same reasoning as if $q_{i_0} \in \mathcal{P}_3$, except for the fact it is now the second rule of Definition A.1.4 which is applied at q_{j_0}, where q_{j_0} the longest common prefix of q_{i_0} and p, and necessarily $R_{i_0-1}|_{q_{j_0}} = \mathsf{dec}(R_l, R_r)$. If it were a proj_i, j_0 would not be the longest common prefix.

Thus the result for any position q_{i_0}, and the existence of such recipes in general. □

A.1.2 Decision for bounded protocols

Here we detail how symbolic traces can be formally linked to the concrete executions of the protocol so as to properly prove Proposition 4.2.3.

First, rather than deal with the usual notion of static equivalence or inclusion (*i.e.* only the direct implications in Definition 2.4.1), we use a variation which we prove to be equivalent in the next lemma.

Lemma A.1.13 (alternative definition of static inclusion)**.** We say that $\phi_1 \sqsubseteq' \phi_2$ if:

- for every ϕ_1-precompact recipe M, if $M\phi_1\downarrow$ is an atom, then $M\phi_2\downarrow$ is an atom,
- for every ϕ_1-precompact recipe M, $M\phi_2\downarrow$ is a message,
- for every ϕ_1-precompact recipe M and N, if $M\phi_1\downarrow = N\phi_1\downarrow$ then $M\phi_2\downarrow = N\phi_2\downarrow$.

and similarly, $\phi_1 \sim' \phi_2$ if $\phi_1 \sqsubseteq' \phi_2$ and $\phi_2 \sqsubseteq' \phi_1$.

Then $\phi_S \sqsubseteq \phi_2$ if, and only if, $\phi_1 \sqsubseteq' \phi_2$.

Proof. We need to show that $\phi_1 \sqsubseteq \phi_2$ if, and only if, $\phi_1 \sqsubseteq' \phi_2$, where $\sqsubseteq$ denotes the static inclusion, *i.e.* only the direct implications in Definition 2.4.1. The implication $\phi_1 \sqsubseteq \phi_2 \Rightarrow \phi_1 \sqsubseteq' \phi_2$ is direct, as this definition only examine fewer tests than the original one and the ability for the attacker to test whether a recipe R reduces to an atom is already ensured by tests of the form $\mathsf{dec}(\mathsf{enc}(\mathsf{w}_1, R)) = \mathsf{w}_1$. So let us consider the converse implication, and assume that

$\phi_1 \sqsubseteq' \phi_2$ and suppose $\phi_1 \not\sqsubseteq \phi_2$. We proceed by induction on recipes, proving our transformation of a single recipe or a pair of recipes strictly decreases the number of constructors (in the single recipe or the sum for pairs). We claim that if we have a witness of static non-inclusion, there is a recipe or a pair of recipes which are destructor-only witnessing that. In the following, M will denote a recipe such that $M\phi_1\downarrow$ is a message but $M\phi_2\downarrow$ is not; M_1 and M_2 will denote two recipes such that $M_i\phi_j\downarrow$ is a message for $i, j \in \{1, 2\}$, $M_1\phi_1\downarrow = M_2\phi_2\downarrow$ but $M_1\phi_2\downarrow \neq M_2\phi_2\downarrow$; and C will be a linear destructor-only context. Let n be the number of constructors in M or the sum of the number of constructors in M_1 and M_2.

- If $n = 0$: M (resp. M_1 and M_2) is destructor-only,
- M contains pairing:
  1. $M = C[\mathsf{dec}(\langle M_1^0, M_2^0\rangle, M_3^0)]$: impossible, as $M\phi_1\downarrow$ would not be a message,
  2. $M = C[\mathsf{proj}_i(\langle M_1^0, M_2^0\rangle)]$: then $M' = C[M_i^0]$ has strictly less than n constructors, is a message in ϕ_1 but still not in ϕ_2,
  3. $M = \langle M_1^0, M_2^0\rangle$: then there exists $i \in \{1, 2\}$ such that M_i^0 is a message in ϕ_1 while not in ϕ_2; and M_i^0 contains strictly less than n constructors,
- M contains encryption:
  1. $M = \mathsf{enc}(M_1^0, M_2^0)$: three subcases must be examined:
     (a) if $M_2^0\phi_2\downarrow$ is a message and is *not* an atom: then M_2^0 contains strictly less than n constructors; and $M_2^0\phi_1\downarrow$ is an atom (as M reduces to a message in ϕ_1),
     (b) else, if $M_2^0\phi_2\downarrow$ is not a message: M_2^0 contains strictly less than that n constructors,
     (c) else, if $M_2^0\phi_2\downarrow$ is an atom: then M_1^0 is a message in ϕ_1 but not in ϕ_2 and contains strictly less than n constructors,
  2. $M = C[\mathsf{proj}_i(\mathsf{enc}(M_1^0, M_2^0))]$: impossible, as $M\phi_1\downarrow$ would not be a message,
  3. $M = C[\mathsf{dec}(\mathsf{enc}(M_1^0, M_2^0), M_3^0)]$: in particular, $M_2^0\phi_1\downarrow = M_3^0\phi_1\downarrow$. If M_2^0 and M_3^0 are both messages in ϕ_2 and $M_2^0\phi_2\downarrow \neq M_2^0\phi_2\downarrow$: the pair (M_2^0, M_3^0) contains $n - 1$ constructors and is a witness of non-inclusion. Else, if $M_i^0\phi_2\downarrow$ is not a message for some $i \in \{2, 3\}$, then it contains strictly less than n constructors. In the remaining case, $M_1^0\phi_2\downarrow$ is not a message, and contains strictly less than n constructors,
- M_1 contains pairing (or symmetrically, M_2 contains pairing), using the same numbering as the case where M did contain pairing:
  1. this case cannot happen,
  2. $C[M_i^0]\phi_j\downarrow = M_1\phi_j\downarrow$ for $j \in \{1, 2\}$, and $C[M_i^0]$ contains strictly less than n constructors,
  3. $M_i^0\phi_2\downarrow = \mathsf{proj}_i(M_1)\phi_2\downarrow$ and $M_1\phi_2\downarrow \neq M_2\phi_2\downarrow$ implies that either $\mathsf{proj}_i(M_2)\phi_2\downarrow$ is not a message, or $M_i^0\phi_2\downarrow \neq \mathsf{proj}_i(M_2)\phi_2\downarrow$; which in both cases counts strictly less than n constructors,
- M_1 contains encryption (or symmetrically, M_2 contains encryption), using the same numbering as the case where M did contain encryption:
  1. $M_2^0\phi_1\downarrow$ and $M_2^0\phi_2\downarrow$ are both messages and atom and $M_1\phi_2\downarrow \neq M_2\phi_2\downarrow$ implies that either $\mathsf{dec}(M_2, M_2^0)\phi_2\downarrow$ is not a message or that $M_1^0\phi_2\downarrow \neq \mathsf{dec}(M_2, M_2^0)\phi_2\downarrow$; which in both cases counts strictly less than n constructors,
  2. this case cannot happen,
  3. $M_1\phi_2\downarrow$ is a message implies $M_2^0\phi_2\downarrow = M_3^0\phi_2\downarrow$ and $M_1^0\phi_2\downarrow$ is a message. Then M_1^0 contains strictly less constructors than M_1, and we get $M_1^0\phi_1\downarrow = M_2\phi_1\downarrow$ while $M_1^0\phi_2\downarrow \neq M_2\phi_2\downarrow$.

At this point, we proved that we can only consider destructor-only witnesses of $\phi_1 \not\sqsubseteq \phi_2$. Suppose now M, M_1 and M_2 are destructor-only: we need to prove they are ϕ_1-precompact, *i.e.* we show they do not reduce to a pair or an encryption with a deducible key.

- if $M\phi_1\downarrow = \langle s, t\rangle$: there exists $i \in \{1, 2\}$ such that $\mathsf{proj}_i(M)$ is still a message in ϕ_1 but not in ϕ_2,
- if $M\phi_1\downarrow = \mathsf{enc}(s, k)$ and k is deducible in ϕ_1, which implies there exists a destructor-only recipe R such that $R\phi_1\downarrow = k$ (consider the $\Downarrow$ normalisation of any recipe reducing to k). $\mathsf{dec}(M, R)$ is a message in ϕ_1 while not in ϕ_2,
- if $M_1\phi_1\downarrow = \langle s, t\rangle = M_2\phi_1\downarrow$: there exists $i \in \{1, 2\}$ such that $\mathsf{proj}_i(M_1)\phi_1\downarrow = \mathsf{proj}_i(M_2)\phi_1\downarrow$ but $\mathsf{proj}_i(M_1)\phi_2\downarrow \neq \mathsf{proj}_i(M_2)\phi_2\downarrow$ or $\mathsf{proj}_i(M_j)\phi_2\downarrow$ is not a message for some $j \in \{1, 2\}$.
- if $M_1\phi_1\downarrow = \mathsf{enc}(s, k) = M_2\phi_1\downarrow$, and k is deducible: as previously, there exists a destructor-only recipe R such that $R\phi_1\downarrow = k$. In that case $\mathsf{dec}(M_1, R)\phi_1\downarrow = \mathsf{dec}(M_2, R)\phi_1\downarrow$. Then either $\mathsf{dec}(M_i, R)\phi_2\downarrow$ is not a message for some $i \in \{1, 2\}$ or $\mathsf{dec}(M_1, R)\phi_2\downarrow \neq \mathsf{dec}(M_2, R)\phi_2\downarrow$.

Note that this last transformation does not introduce any constructor and strictly decreases the normal forms of M, M_1 and M_2 in ϕ_1. Hence, if $\phi_1 \not\sqsubseteq \phi_2$, then $\phi_1 \not\sqsubseteq' \phi_2$. □

Lemmas A.1.14 and A.1.16 provide the link between any concrete second-order trace, as the attacker can build, and the second-order traces which are generated at step 3 of our algorithm. They also provide the relations between concrete and symbolic frames we needed at the beginning of Section A.1.1.

Step 3 in the algorithm induces a renaming ρ. As this renaming is bijective, and only meant to provide concrete traces of P and Q, in the following statements and proofs, we will omit it and refer to the non-renamed trace in the algorithm by (tr_S, ϕ_S), which will then by symbolic.

Lemma A.1.14 (existence of a symbolic trace)**.** For any $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, there exists a *symbolic* second-order trace (tr_S, ϕ_S) of P generated by $\mathcal{A}_{\mathcal{B}}(P, Q)$, a first-order substitution λ_P and a second-order substitution θ such that $\mathsf{tr}\phi\downarrow = \mathsf{tr}_S\phi_S\lambda_P\downarrow$ (in particular: $\phi = \phi_S\lambda_P$), $\mathsf{tr}_S\theta\phi\downarrow = \mathsf{tr}\phi\downarrow$ (the first-order input terms are identical), and such that for every $x \mapsto R_x \in \theta$, R_x is built from the initial knowledge of the attacker and the outputs which preceded the introduction of x in tr_S, $R_x\phi_S\downarrow$ is a message and for every variable x occurring in key position in $\mathsf{tr}_S\phi_S\downarrow$, $R_x\phi_S\downarrow$ is an atom,

Proof. For (tr, ϕ) we can derive the existence of $\mathsf{tr}_0 \in \mathsf{trace}_s(P)$ and a substitution σ such that $\mathsf{tr}\phi\downarrow = \mathsf{tr}_0\sigma$ (Lemma 4.1.1). Using Definition 4.2.3 we get there exists a first-order substitution σ_1 generated by $\mathcal{B}$ applied to tr_0 and a first-order substitution τ such that $\mathsf{tr}_0\sigma = \mathsf{tr}_1\tau$, where $\mathsf{tr}_1 = \mathsf{tr}_0\sigma_1$. (tr_S, ϕ_S) is obtained by lifting tr_1 to second-order with arbitrary valid recipes (*i.e* progressively constructible by the attacker) and storing the outputs of tr_1 in ϕ_S. A fortiori then, $\phi_S\tau = \phi$. Let $\lambda_P = \tau$. To define θ we choose for every $x \in vars(\mathsf{tr}_S)$ a recipe R_x of $x\lambda_P$, then $\theta = \{x \mapsto R_x \text{ for } x \in vars(\mathsf{tr}_S)\}$. Finding such a recipe is always possible thanks to Definition 4.2.3. We can moreover assume $R_x\phi_S\downarrow$ is a symbolic message and if x appears in key position inside $\mathsf{tr}_S\phi_S\downarrow$, then $x\theta\phi_S\downarrow$ is an atom. Indeed, let us order variables in tr_S with their order of apparition (variables introduced simultaneously in an input can be ordered arbitrarily) in tr_S, and define θ_k inductively. If no variables was introduced, $\theta_0 = id$. By induction, suppose θ_k defined up to the k first variables in tr_S such that $y\theta_k\phi_S\downarrow$ is a message if y is one of these k first variables. Let us prove that $R_x\phi_S\downarrow$ can be chosen so that $R_x\phi_S\downarrow$ is a message. Without loss of generality, as keys are atomic, we can choose a recipe R of $x\lambda_P$ such that $R = C[R_1, \ldots, R_m]$, C is a constructor-only context and for every $i \in \{1, \ldots, m\}$, R_i is destructor-only. As both $\downarrow$-reduction and $\Downarrow$-reduction contain only rules with destructors on top, we get that $\mathcal{T}^*_\phi(R) = C[\mathcal{T}^*_\phi(R_1), \ldots, \mathcal{T}^*_\phi(R_m)]$. As $R\phi\downarrow$ and C is constructor-only, for any i, $R_i\phi\downarrow$ is a message. By Lemmas A.1.4 and A.1.7 and Corollary A.1.1, if $\mathcal{T}^*_\phi(R_i) = (R^*_i, \mathcal{R}_i)$, $R_i\phi\downarrow = (R^*_i\theta_k)\phi\downarrow$ and $R^*_i\phi_S\downarrow$ is a message. By induction hypothesis, we deduce $(R^*_i\theta_k)\phi_S\downarrow$ is a message. Taking $R_x = C[R^*_1\theta_k, \ldots, R^*_m\theta_k]$ ensures the result, and $\theta_{k+1} = \theta_k \cup \{x \mapsto R_x\}$. Then, suppose x appears in a key position inside $\mathsf{tr}_S\phi_S\downarrow$ but $x\theta\phi_S\downarrow$

is not an atom. As $\phi = \phi_S\lambda_P$, we would get that $x\lambda_P$ would appear in key position inside $\mathsf{tr}\phi\downarrow$ and $x\lambda_P$ would not be atomic, absurd as $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and keys are atomic. Finally: $(\mathsf{tr}_S\theta)\phi\downarrow = (\mathsf{tr}_S\phi)\downarrow\lambda_P\downarrow$, as $\mathsf{tr}_S\phi = \mathsf{tr}_1$, $(\mathsf{tr}_S\theta)\phi\downarrow = \mathsf{tr}_1\lambda_P = \mathsf{tr}_1\tau = \mathsf{tr}_0\sigma$ and finally $(\mathsf{tr}_S\theta)\phi\downarrow = \mathsf{tr}\phi\downarrow$. □

Lemma A.1.14 aimed at linking any concrete second-order trace of P to the valid second-order trace generated by the algorithm. The next ones, Lemmas A.1.15 and A.1.16 ensure similar properties for (tr_S, ψ_S) when it exists.

Lemma A.1.15 (instances of valid symbolic traces)**.** If (tr_S, ψ_S) is a symbolic second-order trace of Q, then for every valid second-order substitution θ, *i.e.* such that for every $x \mapsto R_x \in \theta$, R_x is built from the initial knowledge of the attacker and the outputs which preceded the introduction of x in tr_S, $R_x\psi_S\downarrow$ is a message and for every variable x occurring in key position in $\mathsf{tr}_S\psi_S\downarrow$, $R_x\psi_S\downarrow$ is an atom, then there exists ψ' such that $(\mathsf{tr}_S\theta, \psi') \in \mathsf{trace}(Q)$ and $\mathsf{tr}_S\theta\psi'\downarrow = \mathsf{tr}_S\psi_S\downarrow(\theta\psi')\downarrow$ and $\psi' = \psi_S(\theta\psi')\downarrow$.

Proof. Let us proceed by induction on n the length of tr_S^n and prove the following statements: $(\mathsf{tr}_S^n\theta, \psi^n) \in \mathsf{trace}(Q)$, $\mathsf{tr}_S^n\theta\psi^n\downarrow = \mathsf{tr}_S^n\psi_S^n\downarrow(\theta\psi^n)\downarrow$, $\psi^n = \psi_S^n(\theta\psi^n)\downarrow$ (which is a particular case of the previous item) and $\sigma^n = \sigma_S^n(\theta\psi^n)\downarrow$; where

$$\sigma^n = mgu((R_1\theta\psi^{n-1}\downarrow, u_1\rho_1), \ldots, (R_m\theta\psi^{n-1}\downarrow, u_m\rho_m))$$
$$\sigma_S^n = mgu((R_1\psi_S^{n-1}\downarrow, u_1\rho_1)\ldots, (R_m\psi_S^{n-1}\downarrow, u_m\rho_m))$$

and:

- tr^n denotes the truncation of tr at length n, m corresponds to the number of inputs in tr^n and ψ^n is the adequate subframe of ψ.
- the u_i are the input patterns of Q filtering the inputs of tr_S,
- ρ_i is the substitution applied to the remaining process after an input (θ in the description of our semantics rules). In particular, $\rho_1 = id$ and

$$\rho_{k+1} = mgu((R_1\theta\psi^{n-1}\downarrow, u_1\rho_1), \ldots, (R_k\theta\psi^{n-1}\downarrow, u_k\rho_k))$$

We can first note that

$$\begin{aligned}\sigma^n &= mgu((R_1\theta\psi^{n-1}\downarrow, u_1\rho_1), \ldots, (R_m\theta\psi^{n-1}\downarrow, u_m\rho_m))\\ &= mgu((R_1\theta\psi^{n-1}\downarrow, u_1), \ldots, (R_m\theta\psi^{n-1}\downarrow, u_m))\end{aligned}$$

by defining of the ρ_i and the unification algorithms for sets of pairs. Similarly,

$$\begin{aligned}\sigma_S^n &= mgu((R_1\psi_S^{n-1}\downarrow, u_1\rho_1)\ldots, (R_m\psi_S^{n-1}\downarrow, u_m\rho_m))\\ &= mgu((R_1\psi_S^{n-1}\downarrow, u_1)\ldots, (R_m\psi_S^{n-1}\downarrow, u_m))\end{aligned}$$

The induction itself:

- n=0: We consider only the initial knowledge of the attacker. As initial frames contain no variables, $\psi^0 = \psi_S^0$; because no input has been made, $\sigma^0 = \sigma_S^0 = \emptyset$; $\mathsf{tr}_S^0\theta\psi^0\downarrow = \mathsf{tr}_S^0\psi_S^0\downarrow(\theta\psi^0)\downarrow$; and finally, $(\mathsf{tr}^0\theta, \psi^0) \in \mathsf{trace}(Q)$ as it has length zero.
- Suppose we get the result up to some n: we make a disjunction on the $n+1$-th action in tr_S.
  - If tr_S^{n+1} ends by an output w (stored in ψ_S^{n+1}) on some channel c. As tr_S is a valid symbolic trace, any variable occurring in $\mathsf{w}\psi_S^n$ was first introduced in former inputs. $(\mathsf{tr}_S^n\theta, \psi^n) \in \mathsf{trace}(Q)$ by induction hypothesis; $\sigma^{n+1} = \sigma^n$ and $\sigma_S^{n+1} = \sigma_S^n$ as there is no new input; and finally, according to the input rule in our semantics, if v_{n+1} is the output pattern of the protocol specification being instantiated, $\mathsf{w}\psi_S^{n+1} = v_{n+1}\sigma_S^n$ and $\mathsf{w}\psi^{n+1} = v_{n+1}\sigma^n$. Hence $\mathsf{w}\psi_S^{n+1}(\theta\psi^n\downarrow) = \mathsf{w}\psi^{n+1}$, and $\psi^{n+1} = \psi_S^{n+1}(\theta\psi^n)\downarrow = \psi_S^{n+1}(\theta\psi^{n+1})\downarrow$ by induction hypothesis (and as the output does not introduce new variables in ψ^{n+1}). And thus $\mathsf{tr}_S^{n+1}\theta\psi^{n+1}\downarrow = \mathsf{tr}_S^{n+1}\psi_S^{n+1}\downarrow(\theta\psi^{n+1})\downarrow$, as the new action of the trace is the output for which the equality has been proved with the new frames ψ_S^{n+1} and ψ^{n+1}.

– If tr_S^{n+1} ends by an output of a channel c: c does not contain any variable and its value only depends on the number of channel outputted so far. As $\mathsf{tr}_S^n\theta\psi^n\downarrow = \mathsf{tr}_S^n\psi_S^n\downarrow(\theta\psi^n)\downarrow$ by induction hypothesis, no substitution is computed in the semantics nor any element added to the frame, we directly derive $\mathsf{tr}_S^{n+1}\theta\psi^{n+1}\downarrow = \mathsf{tr}_S^{n+1}\psi_S^{n+1}\downarrow(\theta\psi^{n+1})\downarrow$, $\psi^{n+1} = \psi^n$, $\sigma^{n+1} = \sigma^n$ (and similarly with their symbolic counterparts).

– If tr_S^{n+1} ends by an input with recipe R_{n+1} on some channel c. According to the input rule semantics, let u_{m+1} be the pattern to be match against R_{m+1} (in tr_S^{n+1}) and against $R_{m+1}\theta$ (in $\mathsf{tr}_S^{n+1}\theta$). Let σ_S^{n+1} (resp. σ^{n+1}) be the substitution introduced by the rule for tr_S^{n+1} (resp. $\mathsf{tr}_S^{n+1}\theta$). We have that:

$$\begin{aligned}\sigma_S^{n+1} &= mgu((R_1\psi_S^n\downarrow, u_1)\ldots,(R_m\psi_S^n\downarrow, u_m),\\ &\qquad (R_{m+1}\psi_S^n\downarrow, u_{m+1}))\\ \sigma^{n+1} &= mgu((R_1\theta\psi^n\downarrow, u_1)\ldots,(R_m\theta\psi^n\downarrow, u_m),\\ &\qquad (R_{m+1}\theta\psi^n\downarrow, u_{m+1}))\end{aligned}$$

These equalities can be rewritten as:

$$\begin{aligned}\sigma_S^{n+1} &= mgu(mgu((R_1\psi_S^n\downarrow, u_1)\ldots,\\ &\quad (R_m\psi_S^n\downarrow, u_m)), (R_{m+1}\psi_S^n\downarrow, u_{m+1}))\\ \sigma^{n+1} &= mgu(mgu((R_1\theta\psi^n\downarrow, u_1)\ldots,\\ &\quad (R_m\theta\psi^n\downarrow, u_m)), (R_{m+1}\theta\psi^n\downarrow, u_{m+1}))\end{aligned}$$

Note that ψ^n is actually equal to ψ^{n-1} on their common domain. Hence we get:

$$\begin{aligned}\sigma_S^{n+1} &= mgu(\sigma_S^n, mgu(R_{m+1}\psi_S^n\downarrow, u_{m+1}))\\ \sigma^{n+1} &= mgu(\sigma^n, mgu(R_{m+1}\theta\psi^n\downarrow, u_{m+1}))\end{aligned}$$

We now need to show that:

$$mgu(R_{m+1}\theta\psi^n\downarrow, u_{m+1}) = mgu(R_{m+1}\psi_S^n\downarrow, u_{m+1})(\theta\psi^n\downarrow)$$

Indeed, $R_{m+1}\theta\psi^n\downarrow = (R_{m+1}\psi^n)\downarrow(\theta\psi^n)\downarrow$ and as $dom(u_{m+1}) \cap dom(\theta) = \emptyset$:

$$\begin{aligned}&mgu((R_{m+1}\psi_S^n)\downarrow(\theta\psi^n)\downarrow, u_{m+1})\\ =\ &mgu(R_{m+1}\psi_S^n\downarrow, u_{m+1})(\theta\psi^n\downarrow)\end{aligned}$$

Hence, as $\sigma^n = \sigma_S^n(\theta\psi^n)\downarrow$, we end up with $\sigma^{n+1} = \sigma_S^{n+1}(\theta\psi^n)\downarrow$, and because the last action of tr_S is an input, $\psi^n = \psi^{n+1}$, we get that $\sigma^{n+1} = \sigma_S^{n+1}(\theta\psi^{n+1})\downarrow$, as intended. Thus $\mathsf{tr}_S^{n+1}\theta\psi^{n+1}\downarrow = \mathsf{tr}_S^{n+1}\psi_S^{n+1}\downarrow(\theta\psi^{n+1})\downarrow$. In particular, $mgu(R_{m+1}\theta\psi^n\downarrow, u_{m+1}) \neq \bot$, so $(\mathsf{tr}_S^{n+1}\theta, \psi^{n+1}) \in \mathsf{trace}(Q)$ and $\psi^{n+1} = \psi_S^{n+1}(\theta\psi^{n+1})\downarrow$.

Thus our induction is complete, and for n equal to the length of tr_S, naming $\psi' = \psi^n$, we prove the desired result. □

Let $\sqsubseteq_k$ denotes the trace inclusion up to length k *i.e.*: $P \sqsubseteq_k Q$ if for any trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ of length (in term of number of actions) lesser than or equal to k, there exits $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ such that $\phi \sim' \psi$.

Lemma A.1.16 (from ϕ to ψ)**.** Let $n \in \mathbb{N}^*$. If $P \sqsubseteq_{n-1} Q$, given $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ of length $k \leq n$, $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$, tr_S and θ as defined in Lemma A.1.14 and ψ_S such that (tr_S, ψ_S) is a symbolic second-order trace, then $\mathsf{tr}\psi\downarrow = \mathsf{tr}_S\psi_S\lambda_Q\downarrow$ and $\psi = \psi_S\lambda_Q$ where $\lambda_Q = (\theta\psi)\downarrow$.

Proof. By definition of θ, for every recipe R of tr and its corresponding recipe R_S in tr_S, $R\phi\downarrow = R_S\theta\phi\downarrow$. As $P \sqsubseteq_{n-1} Q$, it implies $R\psi\downarrow = R_S\theta\psi\downarrow$ (if the last action is an output of a term, the variables are identical, if the last action is an input, their recipes are built on recipes from the frame generated from tr minus its last action, if the last

action is the output of a channel, the channels are identical). By induction on tr and tr_S, we show both traces can share the same execution, thus leading to $\mathsf{tr}_S\theta\psi\downarrow = \mathsf{tr}\psi\downarrow$. From Lemma A.1.15, as there exists ψ' such that $(\mathsf{tr}_S\theta, \psi')$ is also a second-order trace of Q and by induction on tr and tr_S, we show both traces can share the same execution, thus leading to $\psi = \psi'$; and setting $\lambda_Q = \theta\psi\downarrow$, we get that $\psi = \psi_S\lambda_Q$. Hence we finally get that $\mathsf{tr}\psi\downarrow = \mathsf{tr}_S\psi_S\lambda_Q\downarrow$. □

We are now able to prove Proposition 4.2.3 as stated in Section 4.2.3.

Proposition 4.2.3 (completeness)**.** Let P and Q be two bounded protocols such that $P \not\approx Q$. The algorithm $\mathcal{A}_\mathcal{B}$ applied on P and Q returns a minimal (in term of number of actions) witness tr of non-equivalence.

Proof. The proof is by induction on the length of traces. We suppose trace equivalence has been proved up to some length $n-1$ and consider a witness $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ of $P \not\approx_t Q$ of length n. Then we consider for instance the case where $P \not\sqsubseteq_t Q$, *i.e.* (tr, ψ) cannot belong in $\mathsf{trace}(Q)$ and the case where $\phi \not\sim' \psi$. The other cases are handled symmetrically.

From Lemma A.1.14, we get the existence of (tr_S, ϕ_S) a symbolic second-order trace of P along with their substitutions λ_P and θ.

In the first case, suppose that $(\mathsf{tr}, \psi) \notin \mathsf{trace}(Q)$. If for every frame ψ_S, (tr_S, ψ_S) is not a symbolic trace of Q we have a witness. Otherwise let us assume that there exists ψ_S such that (tr_S, ψ_S) is a valid second-order symbolic trace of Q. For every $x \in dom(\theta)$, $x\theta\phi_S\downarrow$ is a message implies that $x\theta\psi_S\downarrow$ is a message, as equivalence between P and Q has been proved up to length $n-1$ (and $x\theta$ must be a recipe built from the frame after tr_S^{-1}). Now, if θ does not satisfy the property that for every variable x occurring in key position in $\mathsf{tr}_S\psi_S\downarrow$, $x\theta\psi_S\downarrow$ is an atom and thus there would exist a variable x such that x appears in key position in $\mathsf{tr}_S\psi_S\downarrow$ while $x\theta\psi_S\downarrow$ is not an atom. Equivalence up to length $n-1$ ensures $x\theta\phi_S\downarrow$ is not an atom. Thus, x should not occur in a key position in $\mathsf{tr}_S\phi_S\downarrow$, meaning $x \in K_{\psi_S} \setminus K_{\phi_S}$. Step 5 in the algorithm would then ensures that if $\lambda = \{^{\langle\omega,\omega\rangle}/_x\}$, $\mathsf{tr}_S\lambda$ is a witness of $P \not\sqsubseteq Q$, as $(\mathsf{tr}_S\lambda, \phi_S\lambda)$ is valid symbolic trace of P but not of Q, as $\langle\omega,\omega\rangle$ would appear in key position in $(\mathsf{tr}_S\lambda)(\psi_S\lambda)\downarrow$. In the following we will then assume θ satisfy this property. Using Lemma A.1.15, we get that $(\mathsf{tr}_S\theta, \psi') \in \mathsf{trace}(Q)$ and $\mathsf{tr}_S\theta\psi'\downarrow = \mathsf{tr}_S\psi_S\downarrow(\theta\psi'\downarrow)$. Because $P \sqsubseteq_{n-1} Q$, there exists ψ'' such that $(\mathsf{tr}^{-1}, \psi'') \in \mathsf{trace}(Q)$ and $\mathsf{tr}_S^{-1}\theta\psi''\downarrow = \mathsf{tr}^{-1}\psi''\downarrow$. Indeed, by definition of θ, for every recipe R of tr and its corresponding recipe R_S in tr_S, $R\phi\downarrow = R_S\theta\phi\downarrow$. As $P \sqsubseteq_{n-1} Q$, it implies $R\psi''\downarrow = R_S\theta\psi''\downarrow$ (if the last action is an output of a term, the variables are identical, if the last action is an input, their recipes are built on recipes from the frame generated from tr minus its last action, if the last action is the output of a channel, the channels are identical). By induction on tr^{-1} and tr_S^{-1}, we show both traces can share the same execution, thus leading to $\mathsf{tr}_S^{-1}\theta\psi''\downarrow = \mathsf{tr}^{-1}\psi''\downarrow$. Moreover after executing $\mathsf{tr}_S^{-1}\theta$ and tr^{-1}, Q ends up in the same configuration (by following the same execution). Then, if tr's last action is an output of a term, because $(\mathsf{tr}_S\theta, \psi') \in \mathsf{trace}(Q)$, there exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$, contradiction. If its last action is an input on some channel c with a recipe R, as we already established that $R_S\theta\psi''\downarrow = R\psi''\downarrow$ and their configurations are identical, there exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$, also reaching a contradiction. If the last action is an output of channel, $(\mathsf{tr}_S, \theta') \in \mathsf{trace}(Q)$ directly implies $(\mathsf{tr}, \psi'') \in \mathsf{trace}(Q)$, which is also a contradiction. Hence (tr_S, ψ_S) is not a valid second-order trace of Q if $(\mathsf{tr}, \psi) \notin \mathsf{trace}(Q)$ for any frame ψ.

Assume $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ for some ψ and $\phi \not\sim' \psi$ for any such ψ. In the second case, we need to prove either that if there is a recipe leading to a message in P but not in Q the algorithm will yield an attack; or that if an equality holds in P but not in Q, the algorithm can derive two equalities which will hold in P and not in Q; or that if a recipe leads to an atom in ϕ but not in ψ, the algorithm witnesses it. As $(\mathsf{tr}_S, \psi_S) \in \mathsf{trace}(Q)$ in this case, from Lemma A.1.16 and because Q is determinate, we can derive the existence of λ_Q such that $\psi = \psi_S\lambda_Q$, which is a required hypothesis of Lemma A.1.10.

In the first subcase, let R be a ϕ-precompact recipe such that $R\phi\downarrow$ is a message but $R\psi\downarrow$ is not. Let us then define $(R^*, \mathcal{R}_1) = \mathcal{T}_\phi^*(R)$ and $(R'^*, \mathcal{R}_2) = \mathcal{T}_\psi^*(R)$. We operate the following disjunction:

1. $\psi_S \not\models \mathcal{R}_1$ or $\phi_S \not\models \mathcal{R}_2$. By Lemma A.1.1, $\phi_S \models \mathcal{R}_1$. Consider for example an equality in $\mathcal{R}_1$ witnessing $\psi_S \not\models \mathcal{R}_1$: this equality is captured by the algorithm at step 6, hence leading to a witness of non-equivalence. The other case is treated symmetrically.

2. else, if $\psi_S \models \mathcal{R}_1$ and $\phi_S \models \mathcal{R}_2$: we can apply Lemma A.1.9, thus $R^* = R'^*$. If $R^*\psi_S\downarrow$ is not a message, as R^* is ϕ_S-compact (because R is ϕ-precompact and of Lemma A.1.8), we have a witness of non-inclusion provided by the algorithm. So let us assume that $R^*\psi_S\downarrow$ is a message. We can now apply Lemma A.1.12. With the same notations, either there exists a ϕ_S-compact recipe R_0 such that $R_0\phi_S\downarrow$ is a message while $R_0\psi_S\downarrow$ is not, in which case we directly get a witness of non-inclusion; or we can consider the next case.

We now deal with the subcase where an equality holds in ϕ but not in ψ. Let R_1 and R_2 be two ϕ-precompact recipes such that $R_1\phi\downarrow = R_2\phi\downarrow$ and $R_1\psi\downarrow \neq R_2\psi\downarrow$. Let us define $(R_1^*, \mathcal{R}_1) = \mathcal{T}_\phi^*(R_1)$, $(R_2^*, \mathcal{R}_2) = \mathcal{T}_\phi^*(R_2)$, $(R_1'^*, \mathcal{R}_1') = \mathcal{T}_\psi^*(R_1)$ and $(R_2'^*, \mathcal{R}_2') = \mathcal{T}_\psi^*(R_2)$. As previously, we can operate the following disjunction:

1. $\psi_S \not\models \mathcal{R}_1$ or $\psi_S \not\models \mathcal{R}_2$ or $\phi_S \not\models \mathcal{R}_1'$ or $\phi_S \not\models \mathcal{R}_2'$: see the previous subcase, we can directly create a witness of non-equivalence.

2. $\psi_S \models \mathcal{R}_1$, $\psi_S \models \mathcal{R}_2$, $\phi_S \models \mathcal{R}_1'$ and $\phi_S \models \mathcal{R}_2'$: we can now apply Lemma A.1.9 twice and get that $R_1'^* = R_1^*$ and $R_2'^* = R_2^*$. From that we apply Lemma A.1.11 and get that there exists σ mgu of two ϕ_S-compact recipes such that such that $R_1^*\phi_S\sigma\downarrow = R_2^*\phi_S\sigma\downarrow$. Thus we can choose $\sigma_1 = \sigma$ at step 2 in the algorithm. And if, for the sake of clarity, we name tr_E, ϕ_E and ψ_E the trace and frames (respectively) introduced at step 3 of the algorithm, it ensures that $R_1^*\phi_E\downarrow = R_2^*\phi_E\downarrow$. From Lemmas A.1.14 and A.1.16, we get new first-order substitutions λ_P' (for (tr_E, ϕ_E)) and λ_Q' (for (tr_E, ψ_E)). From there, we apply the same reasoning as before, as an equality still holds in ϕ but not in ψ, using $\mathcal{T}^*$ and making three separate cases. The first two are identical. In the third one, we now have that we can once again apply Lemma $A.1.9$, followed by Lemma $A.1.11$. In this case, $\sigma = id$ (by construction of ϕ_E), and thus we get an equality $R_1^* = R_2^*$ of compact recipes in ϕ_E and an inequality in ψ_E (Lemma A.1.10).

We finally deal with the case of testing atomicity of a term: suppose there exists R a ϕ-precompact recipe such that $R\phi\downarrow$ is an atom while $R\psi\downarrow$ is not. Let $(R^*, \mathcal{R}_1) = \mathcal{T}_\phi^*(R)$ and $(R'^*, \mathcal{R}_2) = \mathcal{T}_\psi^*(R)$.

- if $\phi_S \not\models \mathcal{R}_2$ or $\psi_S \not\models \mathcal{R}_1$: as before, we get a witness of non-equivalence in our algorithm,

- else, if $\phi_S \models \mathcal{R}_2$ and $\psi_S \models \mathcal{R}_1$, by Lemma A.1.9, $R^* = R'^*$. As R^* is ϕ_S-compact (Lemma A.1.8), $R\phi_S\downarrow$ is not a variable, and in particular $R^*\phi_S\downarrow = R\phi\downarrow$ (by Lemma A.1.4). Hence $R^*\phi_S\downarrow$ is an atom. Now, if $R^*\psi_S\downarrow$ is a variable, $R^* = x$ holds in ψ_S for some variable x but not in ϕ_S, yielding a witness of non static equivalence. Else, if $R^*\psi_S\downarrow$ is a composed term, we get a witness of non static inclusion as $R^*\psi_S\downarrow$ would not be an atom. Finally, if $R\psi_S\downarrow$ is an ground atom, as in that case $R^*\psi_S\downarrow = R\psi\downarrow$ (by Lemma A.1.4), but $R\psi\downarrow$ is not an atom which cannot happen as it would contradict our hypothesis.

In conclusion, we found a witness of length n of $P \not\approx_t Q$ assuming trace equivalence up to length $n-1$: hence we the algorithm derived a shortest witness in terms of number of actions. □

A.2 Proofs of Theorem 4.1.1 and Proposition 4.1.1

In Section A.1, we proved the completeness of the procedure described in Section 4.2. To prove Theorem 4.2.1, and thus Proposition 4.1.1, we still need to prove this algorithm actually preserves types. Once done, we will be able to finally prove Theorem 4.1.1.

A stronger version of Proposition 4.1.1 is actually proven with Theorem 4.2.1: not only there exists such an algorithm, but the algorithm described in Section 4.2.2 does satisfy all the necessary conditions.

Theorem 4.2.1. Let P and Q be two bounded protocols type-compliant w.r.t. $(\mathcal{T}_1, \delta_1)$ and $(\mathcal{T}_2, \delta_2)$ respectively, and such that $P \not\approx Q$. Assume the algorithm $\mathcal{A}_\mathcal{B}$ uses a type-preserving reachability blackbox $\mathcal{B}$ and a well-typed renaming ρ at step 3. Then $\mathcal{A}_\mathcal{B}(P, Q)$ returns a trace tr such that

- either $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ for some ϕ and (tr, ϕ) is pseudo-well-typed w.r.t. $(\mathcal{T}_1, \delta_1)$;
- or $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ for some ψ and (tr, ψ) is pseudo-well-typed w.r.t. $(\mathcal{T}_2, \delta_2)$.

Proof. Assume here the witness of non-equivalence is a trace of P. Because the reachability blackbox preserves types, the trace tr_1 at step 1 is well-typed. At step 2, unification can only happen on a pair (s, t) of ciphertexts with variables: $s, t \in ESt(\mathsf{tr}_1) \subseteq ESt(\mathsf{tr})\sigma_1$. Thus there exists two terms $s', t' \in ESt(P)$ such that $s = s'\sigma_1$ and $t = t'\sigma_1$. Since P is type-compliant w.r.t. $(\mathcal{T}_1, \delta_1)$, $\delta_1(s') = \delta_1(t')$. By definition of a typing system, σ_1 is well-typed and then $\delta_1(t) = \delta_1(s)$ and thus their unifier is well-typed. Hence $\mathsf{tr}_1\sigma_1$ is well-typed, and applying the reachability blackbox and a well-typed renaming leads to (tr, ϕ) being well-typed. If $\mathcal{A}_{\mathcal{B}}(P, Q)$ outputs a witness tr at step 5 and c_0 is the constant replaced by $\langle \omega, \omega \rangle$, as (tr, ϕ) is well-typed and thus $(\mathsf{tr}\lambda, \phi\lambda) \in \mathsf{trace}(P)$, where $\lambda = \{^{\langle \omega, \omega \rangle}/_{\mathsf{c}_0}\}$ is pseudo-well-typed. The symmetric case with traces of Q is handled similarly, as Q is type-compliant w.r.t. $(\mathcal{T}_2, \delta_2)$. □

The proof of Proposition 4.1.1 is then a direct consequence of Theorem 4.2.1.

Theorem 4.1.1. Let P and Q be two determinate protocols type-compliant w.r.t. $(\mathcal{T}_1, \delta_1)$ and $(\mathcal{T}_2, \delta_2)$ respectively. We have that $P \not\approx Q$ if, and only if, there exists a witness of non-equivalence tr such that:

- either $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ for some ϕ and (tr, ϕ) is pseudo-well-typed w.r.t. $(\mathcal{T}_1, \delta_1)$;
- or $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ for some ψ and (tr, ψ) is pseudo-well-typed w.r.t. $(\mathcal{T}_2, \delta_2)$.

Proof. If P or Q contain replication and $P \not\approx_t Q$, there exists a minimal witness tr of length n of this non-equivalence. Consider P' (resp. Q') the unfolding of P where every occurrence of a subprocess $!R$ is replaced by $R|\ldots|R$ $(n+1$ times), with α-renaming to avoid name and variable capture. Because tr is of length n, $\mathcal{A}(P', Q')$ or $\mathcal{A}(Q', P')$ would yield a witness of $P' \not\approx_t Q'$ of length lesser than n, as Proposition 4.2.3 ensures the algorithm returns the shortest witness of non-equivalence. Then, as P' and Q' were unfolded $n+1$ times, this witness is also a witness of $P \not\approx_t Q$. We now need to prove that P' is type-compliant w.r.t. $(\mathcal{T}_1, \delta_1)$ and Q' is type-compliant w.r.t. $(\mathcal{T}_2, \delta_2)$. For instance, let $s', t' \in ESt(P')$ and σ' such that $t'\sigma' = s'\sigma'$. Because P' can be α-renamed so as to use common variable and names with $\mathsf{unfold}^2(P)$, there exist $s, t \in ESt(\mathsf{unfold}^2(P))$, renamings of s' and t' respectively, and σ, renaming of σ', such that $s\sigma = t\sigma$, $\delta_1(s) = \delta_1(s')$ and $\delta_1(t) = \delta_1(t')$ (as α-renaming preserves types). By definition of type-compliant we can conclude that $\delta_1(t) = \delta_1(s)$ and thus P' is type-compliant w.r.t. $(\mathcal{T}_1, \delta_1)$. The same reasoning also applies to Q' w.r.t. $(\mathcal{T}_2, \delta_2)$. Hence P' and Q' are both type-compliant w.r.t. to their respective typing systems and without replication. Hence we only need to deal with the case where P and Q do not use replication. Theorem 4.2.2 and Proposition 4.2.3 ensure that $P \not\approx_t Q$ if, and only if, $\mathcal{A}(P, Q)$ yields a witness, which is well-typed for at least one of these protocols according to Theorem 4.2.1. □

Appendix B

Decidability of trace equivalence for ping-pong protocols

B.1 Undecidability of trace inclusion

The purpose of this section is to establish the following result.

Theorem 7.1.2. The following problem is undecidable.

Input P and Q two protocols in $\mathcal{C}_{\mathsf{pp}}$.

Output Whether P is trace included in Q, i.e. $P \sqsubseteq Q$.

An instance of the PCP over the alphabet A is given by two sets of tiles $U = \{u_i \mid 1 \leq i \leq n\}$ and $V = \{v_i \mid 1 \leq i \leq n\}$ where $u_i, v_i \in A^*$. The problem consists of deciding whether there exists a non-empty sequence $i_1, \ldots, i_p$ over $\{1, \ldots, n\}$ such that $u_{i_1} \ldots u_{i_p} = v_{i_1} \ldots v_{i_p}$.

To prove the undecidability of trace inclusion in $\mathcal{C}_{\mathsf{pp}}$, we show it is possible to encode the Post Correspondence Problem into an inclusion of two protocols of this class. Given a word, one protocol will be meant to unstack the first set of tiles while the other will try as much as possible to unstack the second set of tiles. While an empty word is not "simultaneously" reached by the two processes, their traces appear to be equivalent. Conversely, if a solution to the Post Correspondence Problem does exit, it will lead the second process to react in a distinct way (by stopping its execution), breaking the trace inclusion property.

For each $i \in \{1, \ldots, n\}$, we define two (possibly empty) sets of words over A, namely $W_i \stackrel{\mathsf{def}}{=} A^{|v_i|} \setminus \{v_i\}$, and $W'_i \stackrel{\mathsf{def}}{=} A^0 \cup A^1 \cup \ldots \cup A^{|v_i|-1}$ where $|v_i|$ denote the length of the word v_i.

Example B.1.1. Let $A = \{a, b\}$ and consider the following pairs of tiles (b, ϵ), (b, a), and (a, ba). This instance of PCP admits a solution. Indeed, the non-empty sequence 13 leads to the word $u_1 u_3 = v_1 v_3 = ba$. We have $W_1 = W'_1 = \emptyset$, $W_2 = \{b\}$ and $W'_2 = \{\epsilon\}$, and lastly $W_3 = \{aa, ab, bb\}$ and $W'_3 = \{a, b, \epsilon\}$.

Words in A^* will be represented through nested symmetric encryption with private keys representing their counterparts in A. For the sake of brevity, given a word $u = \alpha_1 \ldots \alpha_p$ of A^*, we denote by:

- $\overline{u}$ the term $\mathsf{senc}(\ldots \mathsf{senc}(\epsilon, \alpha_1, z_1) \ldots, \alpha_p, z_p)$; and
- $\overline{x.u}$ the term $\mathsf{senc}(\ldots \mathsf{senc}(x, \alpha_1, z_1) \ldots, \alpha_p, z_p)$

where $z_1, \ldots, z_p$ are variables of sort rand. Note that if $u = \epsilon$ then $\overline{u} = \epsilon$, and $\overline{x.u} = x$.

Below, $\mathsf{k}_i, \mathsf{k}'_i$ with $i \in \{0, 1, 2, 3\}$ are constants in Σ_0 of sort SymKey, and for each $\alpha \in A$, we denote also by α its counterpart in Σ_0 (constants of sort SymKey). We denote ϵ a constant in Σ_0 of sort msg. These constants are initially unknown by the attacker and actually it is quite easy to see that they will be never revealed. Lastly, c, c_α, c_i, c' with $\alpha \in A$ and $i \in \{1, \ldots, n\}$ are constant symbols of sort channel in $\mathcal{C}h$.

Let P_U and P_V be the following protocols.

$$
\begin{array}{lll}
P_U := & !\,\mathsf{in}(c, \mathsf{start}).\mathsf{new}\ r.\mathsf{out}(c, \mathsf{senc}(\epsilon, \mathsf{k}_0, r)) & (\mathsf{start}) \\
| & !\,\mathsf{in}(c_\alpha, \mathsf{senc}(x, \mathsf{k}_0, z)).\mathsf{new}\ r_1, r_2.\ \mathsf{out}(c_\alpha, \mathsf{senc}(\mathsf{senc}(x, \alpha, r_2), \mathsf{k}_0, r_1)) & (1) \\
| & !\,\mathsf{in}(c_i, \mathsf{senc}(\overline{x.u_i}, \mathsf{k}_0, z)).\mathsf{new}\ r.\mathsf{out}(c_i, \mathsf{senc}(x, \mathsf{k}_1, r)) & (2) \\
| & !\,\mathsf{in}(c_i, \mathsf{senc}(\overline{x.u_i}, \mathsf{k}_1, z)).\mathsf{new}\ r.\mathsf{out}(c_i, \mathsf{senc}(x, \mathsf{k}_1, r)) & (3) \\
| & !\,\mathsf{in}(c', \mathsf{senc}(\epsilon, \mathsf{k}_1, z)).\mathsf{new}\ r.\mathsf{out}(c', \mathsf{senc}(\epsilon, \mathsf{k}_2, r)) & (4)
\end{array}
$$

where i ranges in $\{1, \ldots, n\}$ and α in A.

The branch (start) is the only way to start an execution, then branches (1) are used to build a word $\alpha_1 \ldots \alpha_n$ (that could be a Post word in case we consider a positive instance of PCP). This word will be represented through the term $\mathsf{senc}(\ldots \mathsf{senc}(\epsilon, \alpha_1, r_1), \ldots, \alpha_n, r_n)$ up to the choice of randoms. Then, branches (2) and (3) are used to unstack the different tiles $u_1, \ldots, u_n$. Note that the purpose of having two similar branches (but using different keys) for this task is to ensure that we will unstack at least one tile, and thus the sequence $i_1 \ldots i_p$ of indices is not empty. Then, reaching the empty word when unstacking these tiles will allow us to perform input/output on channel c' (branch (4)).

$$
\begin{array}{lll}
P_V := & !\,\mathsf{in}(c, \mathsf{start}).\mathsf{new}\ r.\mathsf{out}(c, \mathsf{senc}(\epsilon, \mathsf{k}'_0, r)) & (\mathsf{start}) \\
| & !\,\mathsf{in}(c_\alpha, \mathsf{senc}(x, \mathsf{k}'_0, z)).\mathsf{new}\ r_1, r_2.\ \mathsf{out}(c_\alpha, \mathsf{senc}(\mathsf{senc}(x, \alpha, r_2), \mathsf{k}'_0, r_1)) & (1) \\
| & !\,\mathsf{in}(c_i, \mathsf{senc}(\overline{x.v_i}, \mathsf{k}'_0, z)).\mathsf{new}\ r.\mathsf{out}(c_i, \mathsf{senc}(x, \mathsf{k}'_1, r)) & (2') \\
| & !\,\mathsf{in}(c_i, \mathsf{senc}(\overline{x.w}, \mathsf{k}'_0, z)).\mathsf{new}\ r.\mathsf{out}(c_i, \mathsf{senc}(\epsilon, \mathsf{k}'_3, r)) & (2'a) \\
| & !\,\mathsf{in}(c_i, \mathsf{senc}(\overline{w'}, \mathsf{k}'_0, z)).\mathsf{new}\ r.\mathsf{out}(c_i, \mathsf{senc}(\epsilon, \mathsf{k}'_3, r)) & (2'b) \\
| & !\,\mathsf{in}(c_i, \mathsf{senc}(\overline{x.v_i}, \mathsf{k}'_1, z)).\mathsf{new}\ r.\mathsf{out}(c_i, \mathsf{senc}(x, \mathsf{k}'_1, r)) & (3') \\
| & !\,\mathsf{in}(c_i, \mathsf{senc}(\overline{x.w}, \mathsf{k}'_1, z)).\mathsf{new}\ r.\mathsf{out}(c_i, \mathsf{senc}(\epsilon, \mathsf{k}'_3, r)) & (3'a) \\
| & !\,\mathsf{in}(c_i, \mathsf{senc}(\overline{w'}, \mathsf{k}'_1, z)).\mathsf{new}\ r.\mathsf{out}(c_i, \mathsf{senc}(\epsilon, \mathsf{k}'_3, r)) & (3'b) \\
| & !\,\mathsf{in}(c', \mathsf{senc}(\overline{x.\beta}, \mathsf{k}'_1, z)).\mathsf{new}\ r.\mathsf{out}(c', \mathsf{senc}(\epsilon, \mathsf{k}'_2, r)) & (4'a) \\
| & !\,\mathsf{in}(c', \mathsf{senc}(\epsilon, \mathsf{k}'_3, z)).\mathsf{new}\ r.\mathsf{out}(c', \mathsf{senc}(\epsilon, \mathsf{k}'_2, r)) & (4'b) \\
| & !\,\mathsf{in}(c_i, \mathsf{senc}(\epsilon, \mathsf{k}'_3, z)).\mathsf{new}\ r.\mathsf{out}(c_i, \mathsf{senc}(\epsilon, \mathsf{k}'_3, r)) & (5')
\end{array}
$$

where i ranges in $\{1, \ldots, n\}$, α and β in A, and for each $i \in \{1, \ldots, n\}$, w in W_i and w' in W'_i.

The protocol P_V has the same structure as P_U. However, it is more complex since we want P_V to follow the execution of P_U as soon as the execution does not correspond to a solution of the PCP problem. In particular, we do not want P_V to block in case it is not able to unstack a tile v_i. To achieve this, some additional branches are added (namely $(2'a)$ and $(2'b)$, as well as $(3'a)$ and $(3'b)$). Intuitively, those branches are triggered when $(2')$ and $(3')$ can not, and the resulting term is encrypted with a special key k'_3 that will allow P_V to mimic the remaining of the execution using branch $(5')$. Now, regarding the branches on channel c', the idea is to allow P_V to mimic the behaviour of P_U only when the trace tr does not correspond to a solution of the PCP. To achieve this, we allow P_V to follow P_U only when the term given in input on channel c' is not a legal encoding of the empty word. Such a term will go through $(4'a)$ or $(4'b)$.

Note that, on both protocols, the terms that are outputted look like fresh random numbers due to fresh nonces occurring in every output and ignorance of the keys. In other words, the two frames resulting from the execution

of respectively P_U and P_V always remain in static equivalence. Therefore, checking trace equivalence amounts into checking that any execution trace of P_U is a trace of P_V, and conversely.

Lemma B.1.1. The protocols P_U and P_V described above are in $\mathcal{C}_{\mathsf{pp}}$.

Proof. The only non-trivial point is to ensure that condition (2) stated in Definition 7.1.1 is satisfied, *i.e.* to ensure that pattern matching operated by inputs taking place on the same channel is exclusive. Regarding protocol P_U, when two inputs occur on the same channel c_i, we have that the outermost key is different. Regarding protocol P_V, the result also holds thanks to the exclusivity of the pattern matching obtained through a careful definition of sets W_i and W'_i. For instance, note that when $v_i = \epsilon$, $W_i = W'_i = \emptyset$, and thus there is no branch $(2'a)/(2'b)$ (resp. $(3'a)/(3'b)$). □

Proposition B.1.1. Let U/V be an instance of PCP. We have that $P_U \sqsubseteq P_V$ if, and only if, U/V is a negative instance of PCP (*i.e.* an instance with no solution).

Proof. We prove successively the two implications.

($\Rightarrow$) *If U/V is a positive instance of PCP then $P_U \not\sqsubseteq P_V$.* If U/V is a positive instance of PCP, there exists a non-empty sequence $i_1 \dots i_p$ over $\{1, \dots, n\}$ such that $u_{i_1} \cdot \dots \cdot u_{i_p} = v_{i_1} \cdot \dots \cdot v_{i_p}$.

Let $u = \alpha_1 \dots . \alpha_m$ be the resulting word over A. From this word and the sequence $i_1, \dots, i_p$, the attacker playing with P_u can build the term $\mathsf{senc}(\overline{u}, \mathsf{k}_0, r)$ representing the word u with branches (1) and then remove one by one the tiles u_{i_p} to u_{i_1} using (2) and (3). Let tr be the resulting trace of the protocol P_U:

$$\begin{aligned}\mathsf{tr} \stackrel{\mathsf{def}}{=} \mathsf{io}(c, \mathsf{start}, \mathsf{w}_1).\mathsf{io}(c_{\alpha_1}, \mathsf{w}_1, \mathsf{w}_2). \dots . \mathsf{io}(c_{\alpha_m}, \mathsf{w}_m, \mathsf{w}_{m+1}) \\ \mathsf{io}(c_{i_p}, \mathsf{w}_{m+1}, \mathsf{w}_{m+2}) \dots \mathsf{io}(c_{i_1}, \mathsf{w}_{p+m}, \mathsf{w}_{p+m+1}).\mathsf{in}(c', \mathsf{w}_{m+p+1})\end{aligned}$$

where $\mathsf{io}(c, R, \mathsf{w}) \stackrel{\mathsf{def}}{=} \mathsf{in}(c, R).\mathsf{out}(c, \mathsf{w})$.

The trace tr models the fact that, given $\mathsf{senc}(\overline{u}, \mathsf{k}_0, r)$ (stored in w_{m+1}), P_U can remove one by one the tiles u_{i_p} to u_{i_1} to reach the empty word and hence output the message $\mathsf{senc}(\epsilon, \mathsf{k}_1, r)$ (stored in w_{m+p+1}) that can then be accepted as input on c'. In this execution, no equality holds in the resulting frame ϕ, as the attacker ignores the keys that are used to encrypt, and all outputted message use different random seeds; thus all messages look fresh.

We claim that this trace does exist in P_V, *i.e.* there exists no ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(P_V)$. Indeed, the pattern matching operated by P_V is exclusive once the term and the channel is fixed. Thus, P_V has no choice but to remove tiles v_{i_p} to v_{i_1} using $(2')$ and $(3')$ leading to the term $\mathsf{senc}(\epsilon, \mathsf{k}'_1, r)$ (stored in w_{m+p+1}) as $\alpha_1 \dots \alpha_m$ is a Post word. Any other trace would either lead to a mismatch on the channels or an improper filtering in P_V. Then the action $\mathsf{in}(c', \mathsf{w}_{m+p+1})$ will have no counterpart on P_V. So (tr, ϕ) has no equivalent trace in P_V, *i.e.* $P_U \not\sqsubseteq P_V$.

($\Leftarrow$) *If U/V is a negative instance of PCP then $P_U \sqsubseteq P_V$.* Let $(\mathsf{tr}, \phi) \in \mathsf{trace}(P_U)$, we aim at showing that there exists an equivalent trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(P_V)$. Actually, since terms that are outputted by P_U and P_V look like fresh random numbers, we simply have to show that there exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(P_V)$. Two cases can occur for any trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P_U)$:

- tr contains no input on channel c'. In such a case, by construction of P_V, the frame ψ can be built by following the sequence of channels used in tr and choosing the adequate filtering. It is always possible to do so, as the definition of sets W_i and W'_i ensure that every term built by the attacker can be handled on any channel c_i. Note that when the term given in input is of the form $\mathsf{senc}(\epsilon, \mathsf{k}'_3, r)$ for some r, it would be accepted on any channel.

- tr contains an input on channel c'. In such a case, this means that the associated term $\mathsf{senc}(\overline{\alpha_1 \dots \alpha_m}, \mathsf{k}_0, r)$ that has been built using channels c_α with $\alpha \in A$ is a word made of tiles in $\{u_1, \dots, u_n\}$. Indeed, the only way to activate an input on c' is to go through the branches (2) and (3) by unstacking the said tiles. Then, because this particular instance of PCP has no solution, such a word $\alpha_1 \dots \alpha_m$ cannot be a Post word and thus it cannot

be decomposed using tiles in $\{v_1, \ldots, v_n\}$ following the same sequence of indices: because the filtering in P_V is also exhaustive, messages outputted by P_V from a certain point will be either encrypted by k'_3 or will reach the end of the sequence with a term of the form $\mathsf{senc}(u, \mathsf{k}'_1, r)$ with u different from the constant ϵ. Thanks to branches $(4'a)$, $(4'b)$, and $(5')$, P_V will be able to follow P_U.

Hence, for any trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P_U)$ there exists a trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(P_V)$. It remains to show that $\phi \sim \psi$. This is due to the fact that both ϕ and ψ are of the form $\{\mathsf{w}_1 \triangleright \mathsf{senc}(m_1, k_1, r_1), \ldots, \mathsf{w}_n \triangleright \mathsf{senc}(m_n, k_n, r_n)\}$ where the k_i are non deducible and the r_i are "fresh" in the sense that they are all distinct and non deducible. We therefore conclude that $P_U \sqsubseteq P_V$. □

Theorem 7.1.2 directly follows from Proposition B.1.1 and the undecidability of the Post Correspondence Problem.

B.2 Getting rid of the attacker

Lemma 7.2.1. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$, $\mathcal{K}_P$ (resp. $\mathcal{K}_Q$) be the set of deducible constants of sort key that occur in P (resp. Q), if $P \approx Q$ then there exists a unique bijection α from $\mathcal{K}_P$ to $\mathcal{K}_Q$ such that for every trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ there exists a trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ such that for any recipe R and any $\mathsf{k} \in \mathcal{K}_P$:

- $R\phi\!\downarrow$ is of sort s if, and only if, $R\psi\!\downarrow$ is of sort s;

  where $\mathsf{s} \in \{\mathsf{SymKey}, \mathsf{PubKey}, \mathsf{PrivKey}\}$.
- $R\phi\!\downarrow = \mathsf{k}$ if, and only if, $R\psi\!\downarrow = \alpha(\mathsf{k})$;
- $R\phi\!\downarrow = \mathsf{k}^{-1}$ if, and only if, $R\psi\!\downarrow = (\alpha(\mathsf{k}))^{-1}$;

and conversely, for every $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ there exists a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ satisfying the same properties.

Proof. We can describe α as a relation in the following way:

> for every $\mathsf{k} \in \mathcal{K}_P$ of sort s, and every trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and recipe R such that $R\phi\!\downarrow = \mathsf{k}$, we define $\alpha(\mathsf{k}) = R\psi\!\downarrow$ where ψ is the only frame such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$.

The existence of such a frame comes from the fact that $P \approx Q$, whereas its unicity is a consequence of the determinism of protocols in $\mathcal{C}_{\mathsf{pp}}$.

We now need to prove that our definition of α is sound and unambiguous. To do so, we show that:

- *$R\psi\!\downarrow$ is a constant of sort* s. We have that there exists a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ such that $R\phi\!\downarrow = \mathsf{k} \in K_P$. Since $P \approx Q$ and Q is in $\mathcal{C}_{\mathsf{pp}}$, we consider the trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$. By definition of static equivalence, we have that $R\psi\!\downarrow$ is a constant of sort s. Otherwise, we would have that $\mathsf{senc}(\mathsf{start}, R, r_i)\phi \in \mathcal{T}(\Sigma, \mathcal{N})$ whereas $\mathsf{senc}(\mathsf{start}, R, r_i)\psi \notin \mathcal{T}(\Sigma, \mathcal{N})$ if $\mathsf{s} = \mathsf{SymKey}$ (the resulting term is not properly sorted). The same argument applies with raenc and sign for s equal to PubKey and $\mathsf{PrivKey}$ respectively.
- *We have that* $|\mathcal{K}_P| = |\mathcal{K}_Q|$. Suppose *ad absurdum* that, for instance, $|\mathcal{K}_P| < |\mathcal{K}_Q|$. Since every element of $\mathcal{K}_Q$ is deducible (and due to the shape of the protocols under study), there exists $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ such that for all $\mathsf{k} \in \mathcal{K}_Q$, there exists a recipe R_{k} such that $R_{\mathsf{k}}\psi\!\downarrow = \mathsf{k}$. In particular, when $k \neq k'$, we have that $R_k\psi\!\downarrow \neq R_{k'}\psi\!\downarrow$. Since $P \approx Q$, there exists a frame ϕ such that $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$. Thanks to previous item, we know that $R_k\phi\!\downarrow$ (resp $R_{k'}\phi\!\downarrow$) has the same sort as $R_k\psi\!\downarrow$ (resp. $R_{k'}\psi\!\downarrow$), *i.e.* sort key. As $|\mathcal{K}_P| < |\mathcal{K}_Q|$, there exist two distinct keys k and k' such that $R_{\mathsf{k}}\phi\!\downarrow = R_{\mathsf{k}'}\phi\!\downarrow$. Hence ϕ and ψ are not statically equivalent, contradicting the fact that $P \approx Q$. The case where $|\mathcal{K}_Q| < |\mathcal{K}_P|$ can be handle similarly.

- α *is a function.* Suppose there exist a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, a recipe R_i and a corresponding equivalence trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ such that $R_i\phi\downarrow = \mathsf{k}$ and $R_i\psi\downarrow = \mathsf{k}'$; a trace $(\mathsf{tr}', \phi') \in \mathsf{trace}(P)$, a recipe R_j and a corresponding equivalence trace $(\mathsf{tr}', \psi') \in \mathsf{trace}(Q)$ such that $R_j\phi'\downarrow = \mathsf{k}$ but $R_j\psi'\downarrow = \mathsf{k}''$ with $\mathsf{k}' \neq \mathsf{k}''$. Considering the trace made up of the trace tr followed by tr', it is then possible to exhibit a witness of non-equivalence. More precisely, relying on R_i and R_j we can build a test that holds in the resulting frame when executing P, whereas this test will not hold on the frame resulting from the execution of Q.

Now we show that α is an injection, *i.e.* $\alpha(\mathsf{k}) \neq \alpha(\mathsf{k}')$ as soon as k, k' are two distinct elements of $\mathcal{K}_P$. Suppose, as previously, there exist a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, a recipe R_i and a corresponding equivalence trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ such that $R_i\phi\downarrow = \mathsf{k}$ and $R_i\psi\downarrow = \alpha(\mathsf{k})$; a trace $(\mathsf{tr}', \phi') \in \mathsf{trace}(P)$, a recipe R_j and a corresponding equivalence trace $(\mathsf{tr}', \psi') \in \mathsf{trace}(Q)$ such that $R_j\phi'\downarrow = \mathsf{k}'$ but $R_j\psi'\downarrow = \alpha(\mathsf{k})$ with $\mathsf{k} \neq \mathsf{k}'$. Considering the trace made up of the trace tr followed by tr', it is then possible to exhibit a witness of non-equivalence. More precisely, relying on R_i and R_j we can build a test that holds in the frame resulting from the execution of P and that does not hold when executing Q. Thus, we have now prove that α is a bijection.

Note that we have already proved that: $R\phi\downarrow = \mathsf{k}$ if, and only, if $R\psi\downarrow = \alpha(\mathsf{k})$.

To show that α satisfies the last condition (item 3), suppose that $\mathsf{k} \in \mathcal{K}_P$, and $R\phi\downarrow = \mathsf{k}^{-1}$. As previously shown, $R\psi\downarrow = \alpha(\mathsf{k}^{-1})$. We want to prove that $\alpha(\mathsf{k}^{-1}) = (\alpha(\mathsf{k}))^{-1}$. If k is of sort SymKey, the result is obvious as $\mathsf{k}^{-1} = \mathsf{k}$ for any such key. Suppose k is of sort PubKey. We have now that there exists a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and a recipe R' such that $R'\phi\downarrow = \mathsf{k} \in K_P$. Since $P \approx Q$, consider the corresponding equivalence trace $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$. Consider the recipes $R_1 = \mathsf{raenc}(\mathsf{start}, R', n)$ and $R_2 = \mathsf{radec}(R_1, R)$. Then $R_2\phi\downarrow = \mathsf{start}$ and $R_2\psi\downarrow = \mathsf{start}$ if, and only if, $R\psi\downarrow = (R'\psi)^{-1}$. As we have already proved that α preserves sorts, we get that $R_2\psi\downarrow$ is of sort msg if, and only if, $\alpha(\mathsf{k}^{-1}) = R\psi\downarrow = (R'\psi\downarrow)^{-1} = (\alpha(\mathsf{k}))^{-1}$. Hence α is compatible with the inverse function. The same argument can be used if k is of sort $\mathsf{PrivKey}$ with sign and check.

Finally we prove the unicity of such a bijection: suppose there were α' an adequate bijection and $\mathsf{k} \in \mathcal{K}_P$ such that $\alpha(\mathsf{k}) \neq \alpha'(\mathsf{k})$. By definition of α, for every trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and every recipe R such that $R\phi\downarrow = \mathsf{k}$, $\alpha(\mathsf{k}) = R\psi\downarrow$. But as α' satisfy a similar property, we get that $R\psi\downarrow = \alpha'(\mathsf{k})$, contradicting our hypothesis. Hence α is unique. Determinism of P and Q then ensures that traces of P and Q are uniquely matched (as $P \approx Q$), thus guaranteeing the converse part of the Lemma.

□

Lemma 7.2.3. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$ respectively disclosing two sets of keys K and K' as in Lemma 7.2.2. Then $P \approx Q$ if, and only if, $\bar{P} \approx_{\mathsf{fwd}} \bar{Q}$ where:

$$
\begin{aligned}
\bar{P} = P \quad | \quad & \Big|_{\mathsf{k} \in K^{\mathsf{SymKey}}} !\mathsf{in}(c^{\mathsf{senc}}_{\mathsf{k},\alpha(\mathsf{k})}, x).\mathsf{new}\ n.\mathsf{out}(c_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{senc}(x, \mathsf{k}, n)) \\
| \quad & \Big|_{\mathsf{k} \in K^{\mathsf{SymKey}}} !\mathsf{in}(c^{\mathsf{sdec}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{senc}(x, \mathsf{k}, y)).\mathsf{out}(c^{\mathsf{sdec}}_{\mathsf{k},\alpha(\mathsf{k})}, x) \\
| \quad & \Big|_{\mathsf{k} \in K^{\mathsf{PubKey}}} !\mathsf{in}(c^{\mathsf{raenc}}_{\mathsf{k},\alpha(\mathsf{k})}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{raenc}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{raenc}(x, \mathsf{k}, n)) \\
| \quad & \Big|_{\mathsf{k} \in K^{\mathsf{PrivKey}}} !\mathsf{in}(c^{\mathsf{radec}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{raenc}(x, \mathsf{k}, y)).\mathsf{out}(c^{\mathsf{radec}}_{\mathsf{k},\alpha(\mathsf{k})}, x) \\
| \quad & \Big|_{\mathsf{k} \in K^{\mathsf{PrivKey}}} !\mathsf{in}(c^{\mathsf{sign}}_{\mathsf{k},\alpha(\mathsf{k})}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{sign}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{sign}(x, \mathsf{k}, n)) \\
| \quad & \Big|_{\mathsf{k} \in K^{\mathsf{PubKey}}} !\mathsf{in}(c^{\mathsf{check}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{sign}(x, \mathsf{k}, y)).\mathsf{out}(c^{\mathsf{check}}_{\mathsf{k},\alpha(\mathsf{k})}, x)
\end{aligned}
$$

$$\begin{aligned}\bar{Q} = Q \quad | \quad & \Big|_{\mathsf{k}\in K^{\mathsf{SymKey}}} !\mathsf{in}(c^{\mathsf{senc}}_{\mathsf{k},\alpha(\mathsf{k})}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{senc}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{senc}(x,\alpha(\mathsf{k}),n)) \\ | \quad & \Big|_{\mathsf{k}\in K^{\mathsf{SymKey}}} !\mathsf{in}(c^{\mathsf{sdec}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{senc}(x,\alpha(\mathsf{k}),y)).\mathsf{out}(c^{\mathsf{sdec}}_{\mathsf{k},\alpha(\mathsf{k})}, x) \\ | \quad & \Big|_{\mathsf{k}\in K^{\mathsf{PubKey}}} !\mathsf{in}(c^{\mathsf{raenc}}_{\mathsf{k},\alpha(\mathsf{k})}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{raenc}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{raenc}(x,\alpha(\mathsf{k}),n)) \\ | \quad & \Big|_{\mathsf{k}\in K^{\mathsf{PrivKey}}} !\mathsf{in}(c^{\mathsf{radec}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{raenc}(x,\alpha(\mathsf{k}),y)).\mathsf{out}(c^{\mathsf{radec}}_{\mathsf{k},\alpha(\mathsf{k})}, x) \\ | \quad & \Big|_{\mathsf{k}\in K^{\mathsf{PrivKey}}} !\mathsf{in}(c^{\mathsf{sign}}_{\mathsf{k},\alpha(\mathsf{k})}, x).\mathsf{new}\ n.\mathsf{out}(c^{\mathsf{sign}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{sign}(x,\alpha(\mathsf{k}),n)) \\ | \quad & \Big|_{\mathsf{k}\in K^{\mathsf{PubKey}}} !\mathsf{in}(c^{\mathsf{check}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{check}(x,\alpha(\mathsf{k}),y)).\mathsf{out}(c^{\mathsf{check}}_{\mathsf{k},\alpha(\mathsf{k})}, x)\end{aligned}$$

where K^{s} denotes the keys of sort s of K. We call $\mathcal{T}_{\mathsf{oracle}}$ the transformation taking a pair of protocols (P, Q) satisfying the aforementioned condition and returning the pair $(\bar{P}, \bar{Q})$ presently defined.

Proof. Let $\mathcal{K}_P$ (resp. $\mathcal{K}_Q$) be the set of deducible constants of sort key that occur in P (resp. Q). We recall, that, as a consequence of Lemma 7.2.2, we necessarily have that $\mathcal{K}_P \subseteq K$ and $\mathcal{K}_Q \subseteq K'$. Because protocols P and $\bar{P}$ (resp. Q and $\bar{Q}$) disclose all their deducible keys, their exists a trace (tr_0, ϕ_0) of P and $\bar{P}$ (resp. (tr_0, ψ_0) a trace of Q and $\bar{Q}$) defined as follows:

$$\mathsf{tr}_0 = \mathsf{in}(c_{\mathsf{k}_1,\alpha(\mathsf{k}_1)}, \mathsf{start}).\mathsf{out}(c_{\mathsf{k}_1,\alpha(\mathsf{k}_1)}, \mathsf{w}^0_1) \ldots \mathsf{in}(c_{\mathsf{k}_n,\alpha(\mathsf{k}_n)}, \mathsf{start}).\mathsf{out}(c_{\mathsf{k}_n,\alpha(\mathsf{k}_n)}, \mathsf{w}^0_n)$$

for $\mathsf{k}_1, \ldots, \mathsf{k}_n \in \mathcal{K}_P$, and $\phi_0 = \{\mathsf{w}^0_1 \triangleright \mathsf{k}_1, \ldots \mathsf{w}^0_n \triangleright \mathsf{k}_n\}$, and symmetrically for Q and $\bar{Q}$. In the following, we will assume that a trace of P or $\bar{P}$ (resp. of Q or $\bar{Q}$) starts with the prefix tr_0 and contains the frame ϕ_0.

For sake of clarity of the construction explained below, we actually show that:

$$\bar{P} \approx_{\mathsf{fwd}} \bar{Q} \text{ if, and only if } P^+ \approx Q^+$$

where $P^+ = P \mid !\mathsf{in}(c, x).\mathsf{out}(c, x)$ and $Q^+ = Q \mid !\mathsf{in}(c, x).\mathsf{out}(c, x)$ for some fresh channel name c. Then, it is easy to conclude at the expected result relying on the fact that $P \approx Q$ is equivalent to $P^+ \approx Q^+$.

($\Rightarrow$) First, suppose $\bar{P} \not\approx_{\mathsf{fwd}} \bar{Q}$. Assume that there exists $(\mathsf{tr}, \phi) \in \mathsf{trace}_{\mathsf{fwd}}(\bar{P})$ such that there is no equivalent frame ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}_{\mathsf{fwd}}(\bar{Q})$. We define $(\mathsf{tr}', \phi) \in \mathsf{trace}(P^+)$ as follows:

- every sequence $\mathsf{in}(c^{\mathsf{senc}}_{\mathsf{k},\alpha(\mathsf{k})}, R).\mathsf{out}(c^{\mathsf{senc}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{w}')$ in tr is replaced by the sequence $\mathsf{in}(c, \mathsf{senc}(R, \mathsf{w}^0_k, n)).\mathsf{out}(c, \mathsf{w}')$ in tr' where n is a fresh name.
- every sequence $\mathsf{in}(c^{\mathsf{sdec}}_{\mathsf{k},\alpha(\mathsf{k})}, R).\mathsf{out}(c^{\mathsf{sdec}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{w}')$ in tr is replaced by the sequence $\mathsf{in}(c, \mathsf{sdec}(R, \mathsf{w}^0_k)).\mathsf{out}(c, \mathsf{w}')$ in tr'.
- every sequence $\mathsf{in}(c^{\mathsf{raenc}}_{\mathsf{k},\alpha(\mathsf{k})}, R).\mathsf{out}(c^{\mathsf{raenc}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{w}')$ in tr is replaced by the sequence $\mathsf{in}(c, \mathsf{raenc}(R, \mathsf{w}^0_k, n)).\mathsf{out}(c, \mathsf{w}')$ in tr' where n is a fresh name.
- every sequence $\mathsf{in}(c^{\mathsf{radec}}_{\mathsf{k},\alpha(\mathsf{k})}, R).\mathsf{out}(c^{\mathsf{radec}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{w}')$ in tr is replaced by the sequence $\mathsf{in}(c, \mathsf{radec}(R, \mathsf{w}^0_k)).\mathsf{out}(c, \mathsf{w}')$ in tr'.
- every sequence $\mathsf{in}(c^{\mathsf{sign}}_{\mathsf{k},\alpha(\mathsf{k})}, R).\mathsf{out}(c^{\mathsf{sign}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{w}')$ in tr is replaced by the sequence $\mathsf{in}(c, \mathsf{sign}(R, \mathsf{w}^0_k, n)).\mathsf{out}(c, \mathsf{w}')$ in tr' where n is a fresh name.
- every sequence $\mathsf{in}(c^{\mathsf{check}}_{\mathsf{k},\alpha(\mathsf{k})}, R).\mathsf{out}(c^{\mathsf{check}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{w}')$ in tr is replaced by the sequence $\mathsf{in}(c, \mathsf{check}(R, \mathsf{w}^0_k)).\mathsf{out}(c, \mathsf{w}')$ in tr'.

Note that by definition of a trace being in $\mathsf{trace}_{\mathsf{fwd}}(\bar{P})$, we have that R is either a variable w or the constant start. We claim that there exists no frame ψ such that $(\mathsf{tr}', \psi) \in \mathsf{trace}(Q^+)$ with $\phi \sim \psi$. Indeed, because the frame are left unchanged, the input recipes match the same input patterns, and recipes holding true and false keep their truth values. So if such a frame ψ existed, (tr, ψ) would belong to $\mathsf{trace}_{\mathsf{fwd}}(\bar{Q})$ and be equivalent to (tr, ϕ).

($\Leftarrow$) Now, suppose $P \not\approx Q$. We have that $P^+ \not\approx Q^+$, and we can even assume that $P^+ \not\approx^{\mathsf{io}*} Q^+$. We consider a witness of this non-equivalence, *i.e.* a trace tr such that $(\mathsf{tr}, \phi) \in \mathsf{trace}^{\mathsf{io}*}(P^+)$ and for which there exists no equivalent frame ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}^{\mathsf{io}*}(Q^+)$. Actually, we can even assume w.l.o.g. that:

- every input recipe in tr on a channel different from c is either a variable w or the constant start;
- every input recipe in tr on channel c involves at most one function symbol in Σ_{pub};
- $\phi \not\sim_{\mathsf{fwd}} \psi$, *i.e.* we consider recipes that are either variables or the constant start.

We consider the shortest trace $(\mathsf{tr}, \phi) \in \mathsf{trace}^{\mathsf{io}*}(P)$, in terms of number of transitions, such that there is no equivalent frame ψ satisfying $(\mathsf{tr}, \psi) \in \mathsf{trace}^{\mathsf{io}*}(Q)$, and for which all the requirements listed above are satisfied.

Through recipes of the form $\mathsf{senc}(u, v, w)$ on channel c, the attacker has the ability to use the same random seed more that once. Let us first show that we can always assume tr uses nonces at most once. If it is not the case, we build a new trace $(\tilde{\mathsf{tr}}, \tilde{\phi})$, such that ϕ is statically equivalent to $\tilde{\phi}$ for which it is the case.

First, if we consider the case where there exists no ψ such that $(\mathsf{tr}, \psi) \notin \mathsf{trace}^{\mathsf{io}*}(Q)$. Because random seeds are not filtered in protocols of $\mathcal{C}_{\mathsf{pp}}$ (every input pattern contains distinct variables as third argument), we can rename some occurrences of the random seeds of the attacker (*i.e.* the random seeds appearing in the recipes on channel c) by fresh random seeds without changing the status of the trace (*i.e.* the fact that the trace is executable or not). Given tr_ρ such a trace obtained by renaming, we have that $(\mathsf{tr}_\rho, \phi_\rho) \in \mathsf{trace}^{\mathsf{io}*}(P)$ for some frame ϕ_ρ whereas $(\mathsf{tr}_\rho, \psi_\rho) \notin \mathsf{trace}^{\mathsf{io}*}(Q)$ for any frame ψ_ρ. And it particular, if we choose tr_ρ such that there are no two identical nonces in its image, we get a witness of non-equivalence with pairwise distinct random seeds for the attacker.

Now, we consider the case where $(\mathsf{tr}, \psi) \in \mathsf{trace}^{\mathsf{io}*}(Q)$ but $\phi \not\sim_{\mathsf{fwd}} \psi$. Suppose r is a random seed which appears twice in tr, in two contexts $\mathsf{f}(\mathsf{w}_i, \mathsf{w}_j, r)$ and $\mathsf{f}(\mathsf{w}'_i, \mathsf{w}'_j, r)$ for some $\mathsf{f} \in \Sigma_{\mathsf{pub}}$ with $\mathsf{w}_i\phi = \mathsf{w}'_i\phi$ and $\mathsf{w}_j\phi = \mathsf{w}'_j\phi$. Because tr is a minimal witness of non-equivalence, $\phi_{-1} \sim_{\mathsf{fwd}} \psi_{-1}$ where ϕ_{-1} (resp. ψ_{-1}) denotes ϕ (resp. ψ) minus its last element. Consequently we also have that $\mathsf{w}_i\psi = \mathsf{w}'_i\psi$ and $\mathsf{w}_j\psi = \mathsf{w}'_j\psi$, as $\mathsf{w}_i, \mathsf{w}_j, \mathsf{w}'_i, \mathsf{w}'_j \in dom(\phi_{-1})$ (they are used in input recipes). Let w and w' be the corresponding outputs of the recipes $\mathsf{f}(\mathsf{w}_i, \mathsf{w}_j, r)$ and $\mathsf{f}(\mathsf{w}'_i, \mathsf{w}'_j, r)$ and assume w appears before w' in tr: we now have that $\mathsf{w} = \mathsf{w}'$ in both ϕ and ψ, and we can safely replace any occurrence of w' in tr by w. The resulting trace is still a witness of non-equivalence as the substitution replace identical terms in ψ.

Thus, it remains only to consider the case where a random seed appears twice in tr but such that either the function symbol, the plaintext or the keys are different. Formally, consider the two contexts $\mathsf{f}(\mathsf{w}_i, \mathsf{w}_j, r)$ and $\mathsf{g}(\mathsf{w}'_i, \mathsf{w}'_j, r)$ with $\mathsf{f}, \mathsf{g} \in \Sigma_{\mathsf{pub}}$, w and w' their respective outputs variables as before; and either $\mathsf{w}_i\phi \neq \mathsf{w}'_i\phi$, $\mathsf{w}_j\phi \neq \mathsf{w}'_j\phi$ or $\mathsf{f} \neq \mathsf{g}$. Following the same reasoning as before, as $\phi_{-1} \sim_{\mathsf{fwd}} \psi_{-1}$, the same inequality has to hold in ψ. Consider the test $\mathsf{w}_k = \mathsf{w}'_k$ which distinguishes between ϕ and ψ: suppose $\mathsf{w}_k\phi = \mathsf{w}'_k\phi$ but $\mathsf{w}_k\psi \neq \mathsf{w}'_k\psi$. Replacing r by r' in $\mathsf{g}(\mathsf{w}'_i, \mathsf{w}'_j, r)$ will still lead to $\mathsf{w}_k\psi \neq \mathsf{w}'_k\psi$ (after replacement) as no equality between subterms is added. But if $\mathsf{w}_k\phi \neq \mathsf{w}'_k\phi$ (after replacement), it would imply that there were two subterms which became different, and were identical before: but, because the first step already took care of recipes introducing the same random seed twice in the same context, and the protocols in $\mathcal{C}_{\mathsf{pp}}$ cannot use a random seed from an input to use it in another encryption, it is impossible.

Hence, we showed that modifying tr into $\tilde{\mathsf{tr}}$ is a symmetric operation which preserves equalities in the two protocols: identical plaintexts and keys in (tr, ϕ) correspond to identical plaintexts and keys in (tr, ψ), whereas adding fresh nonces does not create any equality in $\tilde{\phi}$ or $\tilde{\psi}$. If (tr, ϕ) does not have any equivalent trace in Q, neither has $(\tilde{\mathsf{tr}}, \tilde{\phi})$. If there exists no frame ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$, then there will exist no frame $\tilde{\psi}$ such that $(\tilde{\mathsf{tr}}, \tilde{\psi}) \in \mathsf{trace}(Q)$ as input filtering is not affected by our transformation. Else, if there exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ but ϕ and ψ are not statically equivalent, because our transformation preserves the terms in the frame, any pair of recipes which

distinguishes between the two of them, will distinguish $\tilde{\phi}$ and $\tilde{\psi}$. So we can always assume than the random seeds occurring in the recipes $\mathsf{f}(u, v, w)$ in (tr, ϕ) are distinct.

Let us now define a corresponding trace $(\bar{\mathsf{tr}}, \bar{\phi}) \in \mathsf{trace}_{\mathsf{fwd}}(\bar{P})$.

- each sequence $\mathsf{in}(c_i, R).\mathsf{out}(c_i, \mathsf{w}')$, where $c_i \neq c$, is left unchanged;
- each sequence $\mathsf{in}(c, \mathsf{f}(R, R_k, n)).\mathsf{out}(c, \mathsf{w}')$, where $R_k\phi{\downarrow} = \mathsf{k}$ and $\mathsf{f} \in \{\mathsf{senc}, \mathsf{raenc}, \mathsf{sign}\}$, is replaced by $\mathsf{in}(c^{\mathsf{f}}_{\mathsf{k},\alpha(\mathsf{k})}, R).\mathsf{out}(c^{\mathsf{f}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{w}')$;
- each sequence $\mathsf{in}(c, \mathsf{g}(R, R_k)).\mathsf{out}(c, \mathsf{w}')$, where $R_k\phi{\downarrow} = \mathsf{k}$ and $\mathsf{g} \in \{\mathsf{sdec}, \mathsf{radec}, \mathsf{check}\}$, is replaced by $\mathsf{in}(c^{\mathsf{g}}_{\mathsf{k},\alpha(\mathsf{k})}, R).\mathsf{out}(c^{\mathsf{g}}_{\mathsf{k},\alpha(\mathsf{k})}, \mathsf{w}')$.

Note that each recipe R and R_k above is a variable w or the constant start. The corresponding frame $\bar{\phi}$ is then defined according to our semantics. Since we have assume that the random seed occurring in the recipes in tr are distinct, we have that $\bar{\phi} = \phi$.

Finally, because $(\mathsf{tr}, \phi) \in \mathsf{trace}^{\mathsf{io}*}(P)$ has no equivalent in Q, and the definition of $(\bar{\mathsf{tr}}, \bar{\phi})$ does not alter the filtering on inputs nor equalities between terms in the frame, $(\bar{\mathsf{tr}}, \bar{\phi}) \in \mathsf{trace}_{\mathsf{fwd}}(\bar{P})$ has no equivalent in $\bar{Q}$.

□

B.3 Encoding a protocol into a real-time GPDA

B.3.1 Characterisation of trace equivalence

Lemma B.3.1. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$, if $P \approx_{\mathsf{fwd}} Q$ then for every trace $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ and every $\mathsf{w}, \mathsf{w}' \in dom(\sigma_P)$, if $\mathsf{w}\sigma_P = \mathsf{w}'\sigma_P = \mathsf{c}$ for some constant c, then $\mathsf{w}\sigma_Q = \mathsf{w}'\sigma_Q = \mathsf{c}'$ for some constant c' where σ_Q is the frame such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}(Q)$.

Proof. First, note that the frame σ_Q mentioned in the lemma is unique up to some alpha-renaming of the randoms that occur in σ_P. Thus, the choice of the frame σ_Q does not change anything regarding the result that we want to prove.

Actually, the only non-trivial point to prove is that if $\mathsf{w}\sigma_P = \mathsf{c}$, then $\mathsf{w}\sigma_Q$ is necessarily a constant too. Since protocols in $\mathcal{C}_{\mathsf{pp}}$ have a replication for every branch, consider the trace obtained by "playing twice" the trace tr in P and Q, *i.e.* given $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ with

$$\mathsf{tr} = \mathsf{in}(c_{i_1}, \mathsf{start}).\mathsf{out}(c_{i_1}, \mathsf{w}_1) \dots \mathsf{in}(c_{i_l}, \mathsf{w}_{l-1}).\mathsf{out}(c_{i_l}, \mathsf{w}_l)$$

build $(\mathsf{tr}', \sigma'_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ where:

$$\begin{cases} \mathsf{tr}' \stackrel{\mathsf{def}}{=} \mathsf{tr}.\bar{\mathsf{tr}} \\ \bar{\mathsf{tr}} \stackrel{\mathsf{def}}{=} \mathsf{in}(c_{i_1}, \mathsf{start}).\mathsf{out}(c_{i_1}, \mathsf{w}_{|\phi|+1}) \dots \mathsf{in}(c_{i_l}, \mathsf{w}_{|\phi|+l}).\mathsf{out}(c_{i_l}, \mathsf{w}_{|\phi|+l}) \end{cases}$$

where every occurrence of start in tr is kept in $\bar{\mathsf{tr}}$ but occurrences of w_k are replaced by $\mathsf{w}_{|\sigma_P|+k}$, $|\sigma_P|$ being the cardinal of $dom(\sigma_P)$; and $\mathsf{tr}.\bar{\mathsf{tr}}$ denotes the concatenation of the two sequences of labels, which is a valid trace, *i.e.* $(\mathsf{tr}', \sigma'_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$. We get symmetrically $(\mathsf{tr}', \sigma'_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$. In particular, there exists $\mathsf{w}_* \in dom(\sigma'_P)$ with $l < *$ such that $\mathsf{w}\sigma'_P = \mathsf{w}_*\sigma'_P = \mathsf{c}$ and the test $\mathsf{w} = \mathsf{w}_*$ is disjoint, *i.e.* $\mathsf{seq}_{\mathsf{tr}'}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}'}(\mathsf{w}_*)$ share no common prefix. As $P \approx_{\mathsf{fwd}} Q$, necessarily $\mathsf{w}\sigma'_Q = \mathsf{w}_*\sigma'_Q$. Now, because the test is disjoint, $\mathsf{w}\sigma'_Q$ and $\mathsf{w}_*\sigma'_Q$ could not share any random nonces. Hence, $\mathsf{w}\sigma_Q$ is a constant. □

Lemma B.3.2. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$ such that $P \approx_{\mathsf{fwd}} Q$. For every trace $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$, every $\mathsf{w}, \mathsf{w}' \in dom(\sigma_P)$ such that the test $\mathsf{w} = \mathsf{w}'$ is σ_P-valid, σ_P-guarded, and pulled-up in (tr, σ_P), we have that $\mathsf{w} = \mathsf{w}'$ is σ_Q-valid, σ_Q-guarded, and pulled-up in (tr, σ_Q) where σ_Q is the frame such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$

Proof. First, note that the frame σ_Q mentioned in the lemma is unique up to some alpha-renaming of the randoms that occur in σ_P. Thus, the choice of the frame σ_Q does not change anything regarding the result that we want to prove.

The only non-trivial point to prove is that if the test $\mathsf{w} = \mathsf{w}'$ is σ_P-valid, σ_P-guarded, and pulled-up in (tr, σ_P) then it is also σ_Q-guarded and pulled-up in (tr, σ_Q). Note that it is necessarily σ_Q-valid since $P \approx_{\mathsf{fwd}} Q$. Actually, we can still assume that the test $\mathsf{w} = \mathsf{w}'$ is σ_Q-guarded (it would otherwise contradict Lemma B.3.1).

Let $\mathsf{pref} = \mathsf{io}(c_{i_0}, \mathsf{start}, \mathsf{w}_{j_0}) \dots \mathsf{io}(c_{i_p}, \mathsf{w}_{j_{p-1}}, \mathsf{w}_{j_p})$ be the maximal common prefix of $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$. Now, it remains to show that $\mathsf{w} = \mathsf{w}'$ is pulled-up in (tr, σ_Q), *i.e.* $\mathsf{w}\sigma_Q$ does not occur as a subterm in $\mathsf{w}_{j_{-1}}\sigma_Q, \mathsf{w}_{j_0}\sigma_Q, \dots \mathsf{w}_{j_{p-1}}\sigma_Q$ where $\mathsf{w}_{j_{-1}}\sigma_Q = \mathsf{start}$.

Assume that this is not the case, we will show that there exists a trace $(\mathsf{tr}^*, \sigma_Q^*) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$, $\mathsf{w}_*, \mathsf{w}'_* \in dom(\sigma_Q^*)$ such that $\mathsf{w}_*\sigma_Q^* = \mathsf{w}'_*\sigma_Q^*$ whereas $\mathsf{w}_*\sigma_P^* \neq \mathsf{w}'_*\sigma_P^*$, where σ_P^* is the frame such that $(\mathsf{tr}^*, \sigma_P^*) \in \mathsf{trace}_{\mathsf{fwd}}(P)$. Note that such a frame necessarily exists since otherwise it trivially contradicts our hypothesis.

Let $p' \in \{0, \dots, p-1\}$ be the smallest index such that $\mathsf{w}\sigma_Q$ occurs as a subterm in $\mathsf{w}_{j_{p'}}\sigma_Q$. We have that:

$$\mathsf{pref} = s_1.\mathsf{io}(c_{i_{p'}}, \mathsf{w}_{j_{p'-1}}, \mathsf{w}_{j_{p'}}).s_2 \quad \text{and} \quad \begin{cases} \mathsf{seq}_{\mathsf{tr}}(\mathsf{w}) &= \mathsf{pref}.s_3 \\ \mathsf{seq}_{\mathsf{tr}}(\mathsf{w}') &= \mathsf{pref}.s'_3 \end{cases}$$

for some sequence s_1, s_2, s_3, and s'_3.

From these sequences we can define $(\mathsf{tr}^*, \sigma_Q^*)$ with $\mathsf{tr}^* = \mathsf{tr}.\bar{\mathsf{tr}}$. Intuitively, the trace $\bar{\mathsf{tr}}$ is obtained relying on the sequence of channels as indicated in the sequence $s_2.s'_3$ using systematically the last generated recipe to feed the following input, and $\mathsf{w}_{j_{p'}}$ to start. More precisely, assuming that

$$\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}') = s_1.\mathsf{io}(c_{i_{p'}}, \mathsf{w}_{j_{p'-1}}, \mathsf{w}_{j_{p'}}).\mathsf{io}(c_{k_1}, \mathsf{w}_{j_{p'}}, \mathsf{w}_{l_1}).\mathsf{io}(c_{k_2}, \mathsf{w}_{l_1}, \mathsf{w}_{l_2}) \dots \mathsf{io}(c_{k_\ell}, \mathsf{w}_{l_{\ell-1}}, \mathsf{w}_{l_\ell})$$

we have that:

$$\bar{\mathsf{tr}} = \mathsf{io}(c_{k_1}, \mathsf{w}_{j_{p'}}, \mathsf{w}_{|\sigma_P|+1}).\mathsf{io}(c_{k_2}, \mathsf{w}_{|\sigma_P|+1}, \mathsf{w}_{|\sigma_P|+2}) \dots \mathsf{io}(c_{k_\ell}, \mathsf{w}_{|\sigma_P|+l-1}, \mathsf{w}_{|\sigma_P|+\ell})$$

and σ_Q^* defined as expected relying on our semantics. Let $\mathsf{w}_* = \mathsf{w}$ and $\mathsf{w}'_* = \mathsf{w}_{|\sigma_P|+\ell}$. We can now show that:

1. *The test $\mathsf{w}_* = \mathsf{w}'_*$ is σ_Q^*-valid and σ_Q^*-guarded.* Indeed, by definition of tr^*, $\mathsf{w}'_*\sigma_Q^*$ and $\mathsf{w}'\sigma_Q$ are already equal up to a renaming of random seeds, as the channel components of $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$ and $\mathsf{seq}_{\mathsf{tr}_*}(\mathsf{w}'_*)$ match. As $\mathsf{w}_*\sigma_Q^* = \mathsf{w}\sigma_Q = \mathsf{w}'\sigma_Q$, $\mathsf{w}_*\sigma_Q^*$ and $\mathsf{w}'_*\sigma_Q^*$ are equal up to a renaming of their random seeds. Lastly, we have that $\mathsf{w}_*\sigma_Q^*$ and $\mathsf{w}'_*\sigma_Q^*$ are both subterms of $\mathsf{w}_{j_{p'}}\sigma_Q^*$, hence $\mathsf{w}_*\sigma_Q^* = \mathsf{w}'_*\sigma_Q^*$.
2. *The test $\mathsf{w}_* = \mathsf{w}'_*$ is pulled-up in* $(\mathsf{tr}^*, \sigma_Q^*)$. This is by construction of tr^*.

Finally, as $P \approx_{\mathsf{fwd}} Q$, there exists σ_P^* such that $(\mathsf{tr}^*, \sigma_P^*) \in \mathsf{trace}_{\mathsf{fwd}}(P)$. But now $\mathsf{w}_* = \mathsf{w}'_*$ is σ_Q^*-valid, σ_Q^*-guarded and pulled-up in $(\mathsf{tr}^*, \sigma_Q^*)$. Moreover, we are now in a situation where the top-level random seeds of $\mathsf{w}_*\sigma_P^*$ and $\mathsf{w}'_*\sigma_P^*$ are generated outside the common prefix of $\mathsf{seq}_{\mathsf{tr}^*}(\mathsf{w}_*)$ and $\mathsf{seq}_{\mathsf{tr}^*}(\mathsf{w}'_*)$, and thus it implies that $\mathsf{w}_*\sigma_P^* \neq \mathsf{w}'_*\sigma_P^*$, contradicting the equivalence $P \approx_{\mathsf{fwd}} Q$. □

Lemma 7.3.1. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$, then $P \approx_{\mathsf{fwd}} Q$ if, and only if, the following four conditions are satisfied:

- CONST$_P$: For all $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$, there exists a frame σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$ and for every $\mathsf{w}, \mathsf{w}' \in dom(\sigma_P)$ and for every constant $\mathsf{c} \in \Sigma_0 \cup \{\mathsf{start}\}$, $\mathsf{w}\sigma_P = \mathsf{w}'\sigma_Q = \mathsf{c}$ if, and only if, there exists a constant $\mathsf{c}' \in \Sigma_0 \cup \{\mathsf{start}\}$ such that $\mathsf{w}\sigma_Q = \mathsf{w}'\sigma_Q = \mathsf{c}'$.
- CONST$_Q$: Similarly swapping the roles of P and Q.
- GUARDED$_P$: For all $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$, there exists a frame σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$ and every test that is σ_P-valid, σ_P-guarded, and pulled-up in (tr, σ_P) is also σ_Q-valid, σ_Q-guarded, and pulled-up in (tr, σ_Q).

- GUARDED$_Q$: Similarly swapping the roles of P and Q.

Proof. We prove the two directions separately.

($\Rightarrow$) This implication is a direct consequence of Lemma B.3.1 and Lemma B.3.2.

($\Leftarrow$) Suppose that $P \not\approx_{\mathsf{fwd}} Q$. This means that there exists for instance $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ such that either there exists no frame σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$, in which case conditions CONST$_P$ and GUARDED$_P$ fail, or σ_Q is indeed defined and there exists a test $\mathsf{w} = \mathsf{w}'$ such that $\mathsf{w}\sigma_P = \mathsf{w}'\sigma_P$ but $\mathsf{w}\sigma_Q \neq \mathsf{w}'\sigma_Q$ (or the converse). Let us assume that $\mathsf{w}\sigma_P = \mathsf{w}'\sigma_P$ but $\mathsf{w}\sigma_Q \neq \mathsf{w}'\sigma_Q$.

If $\mathsf{w}\sigma_P = \mathsf{w}'\sigma_P = \mathsf{c}$ for some constant c, then condition CONST$_P$ is false.

Otherwise, we have that the test $\mathsf{w} = \mathsf{w}'$ is σ_P-valid and σ_P-guarded. From tr and $\mathsf{w} = \mathsf{w}'$, we will build a new trace $(\mathsf{tr}^*, \sigma_P^*)$ and a new test $\mathsf{w}_* = \mathsf{w}'_*$ which is σ_P^*-valid, σ_P^*-guarded, and also pulled-up in $(\mathsf{tr}^*, \sigma_P^*)$. Actually, we proceed as in the proof of the previous lemma.

Let $\mathsf{pref} = \mathsf{io}(c_{i_0}, \mathsf{start}, \mathsf{w}_{j_0}) \dots \mathsf{io}(c_{i_p}, \mathsf{w}_{j_{p-1}}, \mathsf{w}_{j_p})$ be the maximal common prefix of $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$. Let $p' \in \{0, \dots, p-1\}$ be the smallest index such that $\mathsf{w}\sigma_P$ occurs as a subterm in $\mathsf{w}_{j_{p'}}\sigma_P$. Note that if this index does not exist then the test $\mathsf{w} = \mathsf{w}'$ is already pulled-up in (tr, σ_P) and we are done.

We have that:

$$\mathsf{pref} = s_1.\mathsf{io}(c_{i_{p'}}, \mathsf{w}_{j_{p'-1}}, \mathsf{w}_{j_{p'}}).s_2 \quad \text{and} \quad \begin{cases} \mathsf{seq}_{\mathsf{tr}}(\mathsf{w}) &= \mathsf{pref}.s_3 \\ \mathsf{seq}_{\mathsf{tr}}(\mathsf{w}') &= \mathsf{pref}.s'_3 \end{cases}$$

for some sequence s_1, s_2, s_3, and s'_3.

From these sequences, we can define $(\mathsf{tr}^*, \sigma_P^*)$ with $\mathsf{tr}^* = \mathsf{tr}, \bar{\mathsf{tr}}$. Intuitively, the trace $\bar{\mathsf{tr}}$ is obtained relying on the sequence of channels as indicated in the sequence $s_2.s'_3$ using systematically the last generated recipe to feed the following input, and $\mathsf{w}_{j_{p'}}$ to start. More precisely, assuming that

$$\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}') = s_1.\mathsf{io}(c_{i_{p'}}, \mathsf{w}_{j_{p'-1}}, \mathsf{w}_{j_{p'}}).\mathsf{io}(c_{k_1}, \mathsf{w}_{j_{p'}}, \mathsf{w}_{l_1}).\mathsf{io}(c_{k_2}, \mathsf{w}_{l_1}, \mathsf{w}_{l_2}) \dots \mathsf{io}(c_{k_\ell}, \mathsf{w}_{l_{\ell-1}}, \mathsf{w}_{l_\ell})$$

we have that:

$$\bar{\mathsf{tr}} = \mathsf{io}(c_{k_1}, \mathsf{w}_{j_{p'}}, \mathsf{w}_{|\sigma_P|+1}).\mathsf{io}(c_{k_2}, \mathsf{w}_{|\sigma_P|+1}, \mathsf{w}_{|\sigma_P|+2}) \dots \mathsf{io}(c_{k_\ell}, \mathsf{w}_{|\sigma_P|+l-1}, \mathsf{w}_{|\sigma_P|+\ell})$$

and σ_P^* defined as expected relying on our semantics. Let $\mathsf{w}_* = \mathsf{w}$ and $\mathsf{w}'_* = \mathsf{w}_{|\sigma_P|+\ell}$.

Now, either there exists no frame σ_Q^* such that $(\mathsf{tr}^*, \sigma_Q^*) \in \mathsf{trace}(Q)$, in which case condition GUARDED$_P$ fails obviously, or such a frame exists. In this case, by construction of tr^*, we have that the test $\mathsf{w}_* = \mathsf{w}'_*$ is σ_P^*-valid, σ_P^*-guarded, and pulled-up in $(\mathsf{tr}^*, \sigma_P^*)$.

In order to conclude, it remains to show that $\mathsf{w}_*\sigma_Q^* \neq \mathsf{w}'_*\sigma_Q^*$. We already know that $\mathsf{w}\sigma_Q = \mathsf{w}_*\sigma_Q^*$. Suppose *ad absurdum* that $\mathsf{w}_*\sigma_Q^* = \mathsf{w}'_*\sigma_Q^*$. Because the sequences of channels that occur in $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$ and $\mathsf{seq}_{\mathsf{tr}^*}(\mathsf{w}'_*)$ are the same, $\mathsf{w}'\sigma_Q$ and $\mathsf{w}'_*\sigma_Q^*$ are either constant and equal or of the form $\mathsf{f}(u, \mathsf{k}, r)$ with $\mathsf{f} \in \{\mathsf{senc}, \mathsf{raenc}, \mathsf{sign}\}$ and equal up to a renaming of their random seeds. In the first case, it is enough to conclude that $\mathsf{w}\sigma_Q = \mathsf{w}'\sigma_Q$, which is absurd. In the second case, $\mathsf{w}_*\sigma_Q^*$ and $\mathsf{w}'_*\sigma_Q^*$ being randomised, must have equal top-level random seeds, implying that this nonce was introduced before $\mathsf{io}(c_{i_{p'}}, \mathsf{w}_{j_{p'-1}}, \mathsf{w}_{j_{p'}})$ in the common prefix of their respective sequences. As the said prefix is also common to w and w' in tr, $\mathsf{w}\sigma_Q$ and $\mathsf{w}'\sigma_Q$ share the same top-level random seed and are thus equal, contradicting our hypothesis. Therefore: $\mathsf{w}_*\sigma_Q^* \neq \mathsf{w}'_*\sigma_Q^*$. Hence GUARDED$_P$ is false.

Finally, if $\mathsf{w}\sigma_Q = \mathsf{w}'\sigma_Q$ but $\mathsf{w}\sigma_P \neq \mathsf{w}'\sigma_P$, conditions CONST$_Q$ and GUARDED$_Q$ will similarly fail. □

B.3.2 From trace equivalence to language equivalence

Lemma 7.3.2. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$, the two real-time GPDA $\mathcal{A}^P_{\text{CONST}}$ and A^Q_{CONST} are such that:

$$P \text{ and } Q \text{ satisfy conditions CONST}_P \text{ and CONST}_Q \text{ iff } \mathcal{L}(\mathcal{A}^P_{\text{CONST}}) = \mathcal{L}(\mathcal{A}^Q_{\text{CONST}}).$$

Proof. We prove the two implications separately.

($\Rightarrow$) Assume that $\mathcal{L}(\mathcal{A}^P_{\text{CONST}}) \neq \mathcal{L}(\mathcal{A}^Q_{\text{CONST}})$, and consider w.l.o.g. a word $u \in \mathcal{L}(\mathcal{A}^P_{\text{CONST}}) \setminus \mathcal{L}(\mathcal{A}^Q_{\text{CONST}})$. We distinguish two cases depending on whether u is accepted in state q_0 or q_f.

Case $u = c_{i_1}.c_{i_2} \dots c_{i_l}$ is accepted in q_0: In such a case, we built (tr, σ_P) as follows:

$$\mathsf{tr} = \quad \mathsf{io}(c_{i_1}, \mathsf{start}, \mathsf{w}_1).\mathsf{io}(c_{i_2}, \mathsf{w}_1, \mathsf{w}_2) \dots \mathsf{io}(c_{i_l}, \mathsf{w}_{l-1}, \mathsf{w}_l)$$

with σ_P the substitution defined uniquely as expected from our semantics. We have that $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ as the transition function δ fully captures input filtering and output of terms for protocols in $\mathcal{C}_{\mathsf{pp}}$. Since $u \notin \mathcal{L}(\mathcal{A}^Q_{\text{CONST}})$, we have that $(\mathsf{tr}, \sigma_Q) \notin \mathsf{trace}_{\mathsf{fwd}}(Q)$ for any substitution σ_Q, and thus the condition CONST_P fails.

Case u is accepted in q_f: In such a case, we also build a trace $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ "corresponding" to u. The construction is a bit more involved. We have that u is of the form $c_{i_1} c_{i_2} \dots c_{i_k} c_{\mathsf{test}} c_{j_1} c_{j_2} \dots c_{j_l} c_{\mathsf{end}}$. Let $\mathsf{tr} = \mathsf{tr}_1.\mathsf{tr}_2$ with tr_1 and tr_2 defined as follows:

- $\mathsf{tr}_1 = \mathsf{io}(c_{i_1}, \mathsf{start}, \mathsf{w}_1).\mathsf{io}(c_{i_2}, \mathsf{w}_1, \mathsf{w}_2) \dots \mathsf{io}(c_{i_k}, \mathsf{w}_{k-1}, \mathsf{w}_k)$;
- $\mathsf{tr}_2 = \mathsf{io}(c_{j_1}, \mathsf{start}, \mathsf{w}_{k+1}).\mathsf{io}(c_{j_2}, \mathsf{w}_{k+1}, \mathsf{w}_{k+2}) \dots \mathsf{io}(c_{j_l}, \mathsf{w}_{k+l-1}, \mathsf{w}_{k+l})$;

and σ_P is defined uniquely as expected from our semantics, as P is deterministic. We have that $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ as the transition function δ fully captures input filtering and output of terms for protocols in $\mathcal{C}_{\mathsf{pp}}$. We can now define $\mathsf{w} = \mathsf{w}_k$ and $\mathsf{w}' = \mathsf{w}_{k+l}$. Because the transitions from q_0 to q_c and then from q_c to q_f for some constant c were possible, we get that $\mathsf{w}\sigma_P = \mathsf{w}'\sigma_P = \mathsf{c}$.

We know that $u = u_1 c_{\mathsf{test}} u_2 c_{\mathsf{end}} \notin \mathcal{L}(\mathcal{A}^Q_{\text{CONST}})$, and we may assume that u_1 and u_2 are both in $\mathcal{L}(\mathcal{A}^Q_{\text{CONST}})$. Indeed, otherwise, this means that there exists no substitution σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$, and thus CONST_P fails, and the result holds. From now one, we assume that there exists σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$.

Now, let $q \xrightarrow{c;\alpha/\beta} q'$ be the first failing transition in the run of u in $\mathcal{A}^Q_{\text{CONST}}$. We distinguish several cases:

1. *Case $q = q_0$ and $q' = q_\mathsf{c}$ for some constant* c. In such a case $\mathsf{w}\sigma_Q \neq \mathsf{c}$ for any constant c, and $\mathsf{w}\sigma_Q$ is thus a guarded term. The condition CONST_P fails.
2. *Case $q = q_\mathsf{c}$ and $q' = q_\mathsf{f}$ for some constant* c. In such a case $\mathsf{w}\sigma_Q = \mathsf{c}$ but $\mathsf{w}\sigma \neq \mathsf{c}$, making CONST_P fail once again.

Hence P and Q do not satisfy CONST_P. Symmetrically, if $u \in \mathcal{L}(\mathcal{A}^Q_{\text{CONST}}) \setminus \mathcal{L}(\mathcal{A}^P_{\text{CONST}})$, the condition CONST_Q will fail.

($\Leftarrow$) If P and Q do not satisfy CONST_P (or CONST_Q), *i.e.* there exists a trace $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ such that:

1. either there exists no σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$;
2. or there exist $\mathsf{w}, \mathsf{w}' \in dom(\sigma_P)$ and a constant c such that $\mathsf{w}\sigma_P = \mathsf{w}'\sigma_P = \mathsf{c}$ but: either $\mathsf{w}\sigma_Q$ is not a constant, or $\mathsf{w}\sigma_Q$ is a constant but $\mathsf{w}\sigma_Q \neq \mathsf{w}'\sigma_Q$.

We consider such a trace of minimal length ℓ.

In the first case, thanks to minimality, we have that $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}_\ell) = \mathsf{tr}$. From tr we build a word $u \in \mathcal{L}(\mathcal{A}^P_{\text{CONST}})$ by extracting the channels that occur in tr keeping the order. Since there does not exist σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$, and the transition function δ of the automaton fully captures input filtering and output of terms for protocols in $\mathcal{C}_{\mathsf{pp}}$, we have that $u \notin \mathcal{L}(\mathcal{A}^Q_{\text{CONST}})$.

In the second case, thanks to minimality, we have that tr is actually made up of all the actions that occur in $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$ (note that these two sequences may share some actions). From tr, we built a word $u = u_1 c_{\mathsf{test}} u_2 c_{\mathsf{end}} \in \mathcal{L}(\mathcal{A}^P_{\text{CONST}})$ as follows:

- u_1 is obtained by extracting the channels that occur in $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ preserving the order; and
- u_2 is obtained by extracting the channels that occur in $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$ preserving the order;.

As the transition function δ fully captures input filtering and output of terms for protocols in $\mathcal{C}_{\mathsf{pp}}$, we get that upon reading c_{test}, $\mathcal{A}^P_{\text{CONST}}$ is in q_0, the transition $q_0 \xrightarrow{c_{\mathsf{test}};\omega\mathsf{c}/\omega} q_{\mathsf{c}}$ is indeed possible as $\mathsf{w}\sigma_P = \mathsf{c}$; and similarly upon reading the c_{end}, $\mathcal{A}^P_{\text{CONST}}$ is in q_{c}, the transition $q_{\mathsf{c}} \xrightarrow{c_{\mathsf{end}};\omega\mathsf{c}/\omega} q_f$ is indeed possible as $\mathsf{w}'\sigma_P = \mathsf{c}$, hence $u \in \mathcal{L}(\mathcal{A}^P_{\text{CONST}})$. What remains to show is that $u \notin \mathcal{L}(\mathcal{A}^Q_{\text{CONST}})$. We distinguish two cases:

- *Case* $\mathsf{w}\sigma_Q$ *is not a constant.* In such a case, no transition $q_0 \xrightarrow{c_{\mathsf{test}};\omega\mathsf{c}/\omega} q_{\mathsf{c}}$ will be possible after u_1
- *Case* $\mathsf{w}\sigma_Q$ *is a constant* c *but* $\mathsf{w}\sigma_Q \neq \mathsf{w}'\sigma_Q$. In such a case, the transition $q_0 \xrightarrow{c_{\mathsf{end}};\omega\mathsf{c}/\omega} q_c$ will not be possible after u_2.

Hence u cannot belong to $\mathcal{L}(\mathcal{A}^Q_{\mathsf{c}})$. This allows us to conclude. □

Lemma 7.3.3. Let P and Q be two protocols in $\mathcal{C}_{\mathsf{pp}}$, the two real-time GPDA $\mathcal{A}^P_{\text{GUARDED}}$ and A^Q_{GUARDED} are such that:

$$P \text{ and } Q \text{ satisfy conditions } \text{GUARDED}_P \text{ and } \text{GUARDED}_Q \text{ iff } \mathcal{L}(\mathcal{A}^P_{\text{GUARDED}}) = \mathcal{L}(\mathcal{A}^Q_{\text{GUARDED}}).$$

Proof. We prove the two directions separately.

($\Rightarrow$) Assume that $\mathcal{L}(\mathcal{A}^P_{\text{GUARDED}}) \neq \mathcal{L}(\mathcal{A}^Q_{\text{GUARDED}})$, and consider w.l.o.g. a word $u \in \mathcal{L}(\mathcal{A}^P_{\text{GUARDED}}) \setminus \mathcal{L}(\mathcal{A}^Q_{\text{GUARDED}})$. We distinguish two cases depending on whether the word u is accepted in state q_0 or q_{f}.

Case $u = c_{i_1}c_{i_2}\dots c_{i_l}$ *is accepted in* q_0*:* In such a case, we built (tr, σ_P) as follows:

$$\mathsf{tr} = \quad \mathsf{io}(c_{i_1}, \mathsf{start}, \mathsf{w}_1).\mathsf{io}(c_{i_2}, \mathsf{w}_1, \mathsf{w}_2)\dots\mathsf{io}(c_{i_l}, \mathsf{w}_{l-1}, \mathsf{w}_l)$$

with σ_P the substitution defined uniquely as expected from our semantics. We have that $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ as the transition function δ fully captures input filtering and output of terms for protocols in $\mathcal{C}_{\mathsf{pp}}$. Since $u \notin \mathcal{L}(\mathcal{A}^Q_{\text{CONST}})$, we have that $(\mathsf{tr}, \sigma_Q) \notin \mathsf{trace}_{\mathsf{fwd}}(Q)$ for any substitution σ_Q, and thus the condition GUARDED_P fails.

Case u *is accepted in* q_{f}*:* In such a case, we also build a trace $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ "corresponding" to u. The construction is a bit more involved. We have that u is of the form: $c_{i_1}c_{i_2}\dots c_{i_k}c^{i_0}_{\mathsf{fork}}c_{j_1}c_{j_2}\dots c_{j_l}c_{\mathsf{test}}c_{p_1}c_{p_2}\dots c_{p_m}c_{\mathsf{end}}$. Let $\mathsf{tr} = \mathsf{tr}_0.\mathsf{tr}_1.\mathsf{tr}_2$ with tr_0, tr_1 and tr_2 defined as follows:

- $\mathsf{tr}_0 = \mathsf{io}(c_{i_1}, \mathsf{start}, \mathsf{w}_1).\mathsf{io}(c_{i_2}, \mathsf{w}_1, \mathsf{w}_2)\dots\mathsf{io}(c_{i_k}, \mathsf{w}_{k-1}, \mathsf{w}_k).\mathsf{io}(c_{i_0}, \mathsf{w}_k, \mathsf{w}_{k+1})$;
- $\mathsf{tr}_1 = \mathsf{io}(c_{j_1}, \mathsf{w}_{k+1}, \mathsf{w}_{k+2}).\mathsf{io}(c_{j_2}, \mathsf{w}_{k+2}, \mathsf{w}_{k+3})\dots\mathsf{io}(c_{j_l}, \mathsf{w}_{k+l}, \mathsf{w}_{k+l+1})$;
- $\mathsf{tr}_2 = \mathsf{io}(c_{p_1}, \mathsf{w}_{k+1}, \mathsf{w}_{k+l+2}).\mathsf{io}(c_{p_2}, \mathsf{w}_{k+l+2}, \mathsf{w}_{k+l+3})\dots\mathsf{io}(c_{p_m}, \mathsf{w}_{k+l+m}, \mathsf{w}_{k+l+m+1})$.

and σ_P is uniquely defined as expected from our semantics, as P is deterministic. We have that $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ as the transition function fully captures input filtering and output of terms for protocols in $\mathcal{C}_{\mathsf{pp}}$. We can now define $\mathsf{w} = \mathsf{w}_{k+l+1}$ and $\mathsf{w}' = \mathsf{w}_{k+l+m+1}$. The test is σ_P-guarded (the index k associated to the stack symbol (fork, k) is indeed strictly positive), σ_P-valid (since the last transition from q_2 to q_{f} requires the stack to be identical to the stack before reading c_{test}), and pulled-up in (tr, σ_P) (since the fork tiles allow us to control the first time the top-level random seed of $\mathsf{w}\sigma_P$ appears in the frame).

We know that $u = u_0c^{i_0}_{\mathsf{fork}}u_1c_{\mathsf{test}}u_2c_{\mathsf{end}} \notin \mathcal{L}(\mathcal{A}^Q_{\text{GUARDED}})$, and we may assume w.l.o.g. that $u_0c_{i_0}u_1$ and $u_0c_{i_0}u_2$ are both in $\mathcal{L}(\mathcal{A}^Q_{\text{GUARDED}})$. Indeed, otherwise this means that there exists no frame σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$, and thus GUARDED_P fails, and the result holds. From now on, we assume that there exists σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$.

Now, let $q \xrightarrow{c;\alpha/\beta} q'$ be the first failing transition in the run of u in $\mathcal{A}^Q_{\text{GUARDED}}$. We distinguish several cases:

1. *Case* $q = q_0$ *and* $q' = q_1$. Since we have already assume that $u_0 c_{i_0} u_1$ is in $\mathcal{L}(\mathcal{A}^Q_{\text{GUARDED}})$, this means that the required transition does not exist because $||v_i^j|| = 0$. In such a case, the test $\mathsf{w} = \mathsf{w}'$ (even if it was σ_Q-valid and σ_Q-guarded) can not be a pulled-up one in (tr, σ_Q). Thus the condition GUARDED_P fails.

2. *Case* $q = q_1$ *and* $q' = q_1$. In such a case, this means that a fork tile cannot be unstacked, meaning that the corresponding test (even if it was σ_Q-valid and σ_Q-guarded) will not be pulled-up in (tr, σ_Q), and GUARDED_P is false.

3. *Case* $q = q_1$ *and* $q' = q_2$. In such a case, the problem occurs due to the fact that the fork tile is not at the top of the stack upon becoming test. The corresponding test $\mathsf{w} = \mathsf{w}'$ will not be σ_Q-valid since $\mathsf{w}\sigma_Q$ will contain a random seed that has been generated after the "forking point", and thus this random seed can not occur in $\mathsf{w}'\sigma_Q$. Thus, the condition GUARDED_P fails.

4. *Case* $q = q_2$ *and* $q' = q_\mathsf{f}$. In such a case, the test tile is not at the top of the stack upon reading the last letter of the word. The test is not σ_Q-valid. The stack at this point, without the test tile, is not identical to the stack before the fork tile turning test, making GUARDED_Q fail.

Hence GUARDED_Q fails as soon as $u \notin \mathcal{L}(\mathcal{A}^Q_{\text{GUARDED}})$.

($\Leftarrow$) If P and Q do not satisfy conditions GUARDED_P (or GUARDED_Q), *i.e.* there exists a trace $(\mathsf{tr}, \sigma_P) \in \mathsf{trace}_{\mathsf{fwd}}(P)$ such that:

1. either there exists no σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$;

2. or (such a σ_Q exists) and there exists $\mathsf{w}, \mathsf{w}' \in dom(\sigma_P)$ such that the test $\mathsf{w} = \mathsf{w}'$ is σ_P-guarded and σ_P-valid, and pulled-up in (tr, σ_P), and

   - either $\mathsf{w} = \mathsf{w}'$ is not σ_Q-valid,
   - or $\mathsf{w} = \mathsf{w}'$ is not σ_Q-guarded,
   - or $\mathsf{w} = \mathsf{w}'$ is not pulled-up in (tr, σ_Q).

We consider such a trace of minimal length ℓ.

In the first case, thanks to the minimality, we have that $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}_\ell) = \mathsf{tr}$. From tr we build a word $u \in \mathcal{L}(\mathcal{A}^P_{\text{GUARDED}})$ by extracting the channels that occur in tr keeping the order. Since there does not exist σ_Q such that $(\mathsf{tr}, \sigma_Q) \in \mathsf{trace}_{\mathsf{fwd}}(Q)$, and the transition function δ of the automaton fully captures input filtering and output of terms for protocols in $\mathcal{C}_{\mathsf{pp}}$, we have that $u \notin \mathcal{L}(\mathcal{A}^Q_{\text{GUARDED}})$.

In the second case, thanks to minimality, we have that tr is actually made up of all the actions that occur in $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ and $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$. These two sequences have a maximal common prefix pref that is not empty. Actually, we have that:

$$\mathsf{pref} = \mathsf{io}(c_{i_1}, \mathsf{start}, \mathsf{w}_1).\mathsf{io}(c_{i_2}, \mathsf{w}_1, \mathsf{w}_2) \ldots \mathsf{io}(c_{i_p}, \mathsf{w}_{p-1}, \mathsf{w}_p) \text{ for some } p \geq 1.$$

From tr, we build a word $u = c_{i_1} c_{i_2} \ldots c_{i_{p-1}} c^{i_p}_{\mathsf{fork}} u_1 c_{\mathsf{test}} u_2 c_{\mathsf{end}} \in \mathcal{L}(\mathcal{A}^P_{\text{GUARDED}})$ as follows:

- u_1 is obtained by extracting the channels that occur in $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w})$ after the prefix $c_{i_1} c_{i_2} \ldots c_{i_{p-1}} c_{i_p}$; and
- u_2 is obtained by extracting the channels that occur in $\mathsf{seq}_{\mathsf{tr}}(\mathsf{w}')$ after the prefix $c_{i_1} c_{i_2} \ldots c_{i_{p-1}} c_{i_p}$.

As the transition function δ fully captures input filtering and output of terms for protocols in $\mathcal{C}_{\mathsf{pp}}$, and since $\mathsf{w} = \mathsf{w}'$ is a test that is σ_P-guarded, σ_P-valid and pulled-up in (tr, σ_P), we get that $u \in \mathcal{L}(\mathcal{A}^P_{\text{GUARDED}})$. What remains to show is that $u \notin \mathcal{L}(\mathcal{A}^Q_{\text{GUARDED}})$. We distinguish two cases:

- *Case* $\mathsf{w} = \mathsf{w}'$ *is not* σ_Q*-valid.* In such a case, even if after reading the first part of u, *i.e.* $c_{i_1} c_{i_2} \ldots c_{i_{p-1}} c^{i_p}_{\mathsf{fork}} u_1 c_{\mathsf{test}}$, we reach q_2, then we will fail to read the remaining of the word to end in q_f.

- *Case* $\mathsf{w} = \mathsf{w}'$ *is* σ_Q*-valid but* $\mathsf{w} = \mathsf{w}'$ *is not* σ_Q*-guarded.* In such a case, this means that $\mathsf{w}\sigma_Q$ is a constant, and the run will stop in q_0 after reading $c_{i_1} c_{i_2} \dots c_{i_{p-1}}$. This comes from the fact that it is not possible to go from q_0 to q_1 adding a tile (fork_i^j, k) with $k = 0$.
- *Case* $\mathsf{w} = \mathsf{w}'$ *is* σ_Q*-valid,* σ_Q*-guarded, but not pulled-up in* (tr, σ_Q). The fact that the test is σ_Q-valid but not pulled-up means that the run will stop in q_1 after reading because of the presence of a tile (fork_i^j, k) in the stack that can not go down anymore.

Hence u cannot belong to $\mathcal{L}(\mathcal{A}^Q_{\text{GUARDED}})$. This allows us to conclude. □

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Analyse automatique de propriétés d'équivalence pour les protocoles cryptographiques

Mots-clefs : sécurité, vérification, protocoles, équivalence, décidabilité

Résumé : À mesure que le nombre d'objets capables de communiquer croît, le besoin de sécuriser leurs interactions également. La conception des protocoles cryptographiques nécessaires pour cela est une tâche notoirement complexe et fréquemment sujette aux erreurs humaines. La vérification formelle de protocoles entend offrir des méthodes automatiques et exactes pour s'assurer de leur sécurité.

Nous nous intéressons en particulier aux méthodes de vérification automatique des propriétés d'équivalence pour de tels protocoles dans le modèle symbolique et pour un nombre non borné de sessions. Les propriétés d'équivalence sont naturellement employées pour s'assurer, par exemple, de l'anonymat du vote électronique ou de la non-traçabilité des passeports électroniques.

Parce que la vérification de propriétés d'équivalence est un problème complexe, nous proposons dans un premier temps deux méthodes pour en simplifier la vérification : tout d'abord une méthode pour supprimer l'utilisation des nonces dans un protocole tout en préservant la correction de la vérification automatique; puis nous démontrons un résultat de typage qui permet de restreindre l'espace de recherche d'attaques sans pour autant affecter le pouvoir de l'attaquant.

Dans un second temps nous exposons trois classes de protocoles pour lesquelles la vérification de l'équivalence dans le modèle symbolique est décidable. Ces classes bénéficient des méthodes de simplification présentées plus tôt et permettent d'étudier automatiquement des protocoles taggués, avec ou sans nonces, ou encore des protocoles ping-pong.

Title : Automated analysis of equivalence properties for cryptographic protocols

Keywords : security, verification, protocols, equivalence, decidability

Abstract : As the number of devices able to communicate grows, so does the need to secure their interactions. The design of cryptographic protocols is a difficult task and prone to human errors. Formal verification of such protocols offers a way to automatically and exactly prove their security.

In particular, we focus on automated verification methods to prove the equivalence of cryptographic protocols for a unbounded number of sessions. This kind of property naturally arises when dealing with the anonymity of electronic voting or the untracability of electronic passports.

Because the verification of equivalence properties is a complex issue, we first propose two methods to simplify it: first we design a transformation on protocols to delete any nonce while maintaining the soundness of equivalence checking; then we prove a typing result which decreases the search space for attacks without affecting the power of the attacker.

Finally, we describe three classes of protocols for which equivalence is decidable in the symbolic model. These classes benefit from the simplication results stated earlier and enable us to automatically analyse tagged protocols with or without nonces, as well as ping-pong protocols.